VISIT…

Algorithms and Computation in Mathematics

Volume 5

Dieter Jungnickel

Graphs, Networks and Algorithms

Third Edition

Springer

Algorithms and Computation in Mathematics · Volume 5

Editors

Arjeh M. Cohen Henri Cohen
David Eisenbud Michael F. Singer Bernd Sturmfels

Dieter Jungnickel

Graphs, Networks and Algorithms

Third Edition

With 209 Figures and 9 Tables

Springer

Dieter Jungnickel

Universität Augsburg
Institut für Mathematik
86135 Augsburg
Germany
E-mail: jungnickel@math.uni-augsburg.de

Library of Congress Control Number: 2007929271

Mathematics Subject Classification (2000): 11-01, 11Y40, 11E12, 11R29

ISSN 1431-1550

ISBN 978-3-540-72779-8 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable for prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media
springer.com
© Springer-Verlag Berlin Heidelberg 2008

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting by the author and SPi using a Springer LaTeX macro package
Cover design: *design & production* GmbH, Heidelberg

Printed on acid-free paper SPIN: 12063185 46/SPi 5 4 3 2 1 0

A mathematician, like a painter or a poet,
is a maker of patterns.
If his patterns are more permanent than theirs,
it is because they are made with ideas.

G. H. Hardy

To my teacher, Prof. Hanfried Lenz

Preface to the Third Edition

The show must go on.
Ira Gershwin

This new third edition has again been thoroughly revised, even though the changes are not as extensive as in the second edition. Of course, the general aims of the book have remained the same.

In particular, I have added some additional material, namely two new sections concerning graphical codes (which provides a less obvious area of application and, as I hope, might also interest the reader in the important field of coding theory) and about two dozen further exercises (as usual, with solutions). I have also discussed and referenced recent developments, especially for the travelling salesman problem, where truly impressive new world records have been achieved. Moreover, the presentation of the material has been improved in quite a few places, most notably in the chapters on shortest paths and colorings. In addition to this, many smaller changes and corrections have been made, and the proof of several theorems have been rewritten to make them more transparent, or more precise.

Again, I thank my students and assistants for their attention and interest as well as the input they provided. Moreover, I am indebted to several readers who alerted me to some (fortunately, more or less minor) problems; and I am, of course, also grateful for the encouraging comments I have received.

Augsburg, May 2007 Dieter Jungnickel

Preface to the Second Edition

Change is inevitable...
Change is constant.
Benjamin Disraeli

When the first printing of this book sold out in a comparatively short time, it was decided to reprint the original edition with only small modifications: I just took the opportunity to correct a handful of minor mistakes and to provide a few updates to the bibliography. In contrast, the new second edition has been thoroughly revised, even though the general aims of the book have remained the same. In particular, I have added some new material, namely a chapter on the network simplex algorithm and a section on the five color theorem; this also necessitated some changes in the previous order of the presentation (so that the numbering differs from that of the first edition, beginning with Chapter 8). In addition to this, numerous smaller changes and corrections have been made and several recent developments have been discussed and referenced. There are also several new exercises.

Again, I thank my students and assistants for their attention and interest as well as the input they provided. Moreover, I am particularly grateful to two colleagues: Prof. Chris Fisher who read the entire manuscript and whose suggestions led to many improvements in the presentation; and Priv.-Doz. Dr. Bernhard Schmidt who let me make use of his lecture notes on the network simplex algorithm.

Augsburg, September 2004 Dieter Jungnickel

Preface to the First Edition

The algorithmic way of life is best.
Hermann Weyl

During the last few decades, combinatorial optimization and graph theory have – as the whole field of combinatorics in general – experienced a particularly fast development. There are various reasons for this fact; one is, for example, that applying combinatorial arguments has become more and more common. However, two developments on the outside of mathematics may have been more important: First, a lot of problems in combinatorial optimization arose directly from everyday practice in engineering and management: determining shortest or most reliable paths in traffic or communication networks, maximal or compatible flows, or shortest tours; planning connections in traffic networks; coordinating projects; solving supply and demand problems. Second, practical instances of those tasks which belong to operations research have become accessible by the development of more and more efficient computer systems. Furthermore, combinatorial optimization problems are also important for complexity theory, an area in the common intersection of mathematics and theoretical computer science which deals with the analysis of algorithms. Combinatorial optimization is a fascinating part of mathematics, and a lot of its fascination – at least for me – comes from its interdisciplinarity and its practical relevance.

The present book focuses mainly on that part of combinatorial optimization which can be formulated and treated by graph theoretical methods; neither the theory of linear programming nor polyhedral combinatorics are considered. Simultaneously, the book gives an introduction into graph theory, where we restrict ourselves to finite graphs. We motivate the problems by practical interpretations wherever possible.[1] Also, we use an algorithmic point of view; that is, we are not content with knowing that an optimal solution exists (this is trivial to see in most cases anyway), but we are mainly interested in the problem of how to find an optimal (or at least almost optimal)

[1] Most of the subjects we treat here are of great importance for practical applications, for example for VLSI layout or for designing traffic or communication networks. We recommend the books [Ber92], [KoLP90], and [Len90].

solution as efficiently as possible. Most of the problems we treat have a *good* algorithmic solution, but we also show how even difficult problems can be treated (for example by approximation algorithms or complete enumeration) using a particular *hard* problem (namely the famous *travelling salesman problem*) as an example. Such techniques are interesting even for problems where it is possible to find an exact solution because they may decrease the amount of calculations needed considerably. In order to be able to judge the quality of algorithms and the degree of difficulty of problems, we introduce the basic ideas of complexity theory (in an informal way) and explain one of the main open problems of modern mathematics (namely the question *P=NP?*). In the first chapters of the book, we will present algorithms in a rather detailed manner but turn to a more concise presentation in later parts. We decided not to include any explicit programs in this book; it should not be too difficult for a reader who is used to writing programs to transfer the given algorithms. Giving programs in any fixed programming language would have meant that the book is likely to be obsolete within a short time; moreover, explicit programs would have obscured the mathematical background of the algorithms. However, we use a structured way of presentation for our algorithms, including special commands based on PASCAL (a rather usual approach). The book contains a lot of exercises and, in the appendix, the solutions or hints for finding the solution. As in any other discipline, combinatorial optimization can be learned best by really working with the material; this is true in particular for understanding the algorithms. Therefore, we urge the reader to work on the exercises seriously (and do the mere calculations as well).

The present book is a translation of a revised version of the third edition of my German text book *Graphen, Netzwerke und Algorithmen*. The translation and the typesetting was done by Dr. Tilla Schade with my collaboration.

The text is based on two courses I gave in the winter term 1984/85 and in the summer term 1985 at the Justus-Liebig-University in Gießen. As the first edition of the book which appeared in 1987 was received quite well, a second edition became necessary in 1990. This second edition was only slightly changed (there were only a few corrections and some additions made, including a further appendix and a number of new references), because it appeared a relatively short time after the first edition. The third edition, however, was completely revised and newly typeset. Besides several corrections and rearrangements, some larger supplements were added and the references brought up to date. The lectures and seminars concerning combinatorial optimization and graph theory that I continued to give regularly (first at the University of Gießen, then since the summer term 1993 at the University of Augsburg) were very helpful here. I used the text presented here repeatedly; I also took it as the basis for a workshop for high school students organized by the *Verein Bildung und Begabung*. This workshop showed that the subjects treated in this book are accessible even to high school students; if motivated sufficiently, they approach the problems with great interest. Moreover, the German edition has been used regularly at various other universities.

I thank my students and assistants and the students who attended the workshop mentioned above for their constant attention and steady interest. Thanks are due, in particular, to Priv.-Doz. Dr. Dirk Hachenberger and Prof. Dr. Alexander Pott who read the entire manuscript of the (German) third edition with critical accuracy; the remaining errors are my responsibility.

Augsburg, May 1998 Dieter Jungnickel

Contents

When we have not what we like,
we must like what we have.
Comte de Bussy-Rabutin

Preface to the Third Edition VII

Preface to the Second Edition IX

Preface to the First Edition XI

1 Basic Graph Theory 1
1.1 Graphs, subgraphs and factors 2
1.2 Paths, cycles, connectedness, trees 5
1.3 Euler tours 13
1.4 Hamiltonian cycles 15
1.5 Planar graphs 21
1.6 Digraphs 25
1.7 An application: Tournaments and leagues 28

2 Algorithms and Complexity 33
2.1 Algorithms 34
2.2 Representing graphs 36
2.3 The algorithm of Hierholzer 39
2.4 How to write down algorithms 41
2.5 The complexity of algorithms 43
2.6 Directed acyclic graphs 46
2.7 NP-complete problems 49
2.8 HC is NP-complete 53

3 Shortest Paths 59
3.1 Shortest paths 59
3.2 Finite metric spaces 61
3.3 Breadth first search and bipartite graphs 63
3.4 Shortest path trees 68
3.5 Bellman's equations and acyclic networks 70

3.6 An application: Scheduling projects 72
3.7 The algorithm of Dijkstra 76
3.8 An application: Train schedules 81
3.9 The algorithm of Floyd and Warshall 84
3.10 Cycles of negative length 89
3.11 Path algebras 90

4 Spanning Trees 97
4.1 Trees and forests 97
4.2 Incidence matrices 99
4.3 Minimal spanning trees 104
4.4 The algorithms of Prim, Kruskal and Boruvka 106
4.5 Maximal spanning trees 113
4.6 Steiner trees 115
4.7 Spanning trees with restrictions 118
4.8 Arborescences and directed Euler tours 121

5 The Greedy Algorithm 127
5.1 The greedy algorithm and matroids 127
5.2 Characterizations of matroids 129
5.3 Matroid duality 135
5.4 The greedy algorithm as an approximation method 137
5.5 Minimization in independence systems 144
5.6 Accessible set systems 148

6 Flows 153
6.1 The theorems of Ford and Fulkerson 153
6.2 The algorithm of Edmonds and Karp 159
6.3 Auxiliary networks and phases 169
6.4 Constructing blocking flows 176
6.5 Zero-one flows 185
6.6 The algorithm of Goldberg and Tarjan 189

7 Combinatorial Applications 209
7.1 Disjoint paths: Menger's theorem 209
7.2 Matchings: König's theorem 213
7.3 Partial transversals: The marriage theorem 218
7.4 Combinatorics of matrices 223
7.5 Dissections: Dilworth's theorem 227
7.6 Parallelisms: Baranyai's theorem 231
7.7 Supply and demand: The Gale-Ryser theorem 234

8 Connectivity and Depth First Search 239
8.1 k-connected graphs 239
8.2 Depth first search 242
8.3 2-connected graphs 245
8.4 Depth first search for digraphs 252
8.5 Strongly connected digraphs 253
8.6 Edge connectivity 258

9 Colorings 261
9.1 Vertex colorings 261
9.2 Comparability graphs and interval graphs 265
9.3 Edge colorings 268
9.4 Cayley graphs 271
9.5 The five color theorem 275

10 Circulations 279
10.1 Circulations and flows 279
10.2 Feasible circulations 282
10.3 Elementary circulations 289
10.4 The algorithm of Klein 295
10.5 The algorithm of Busacker and Gowen 299
10.6 Potentials and ε-optimality 302
10.7 Optimal circulations by successive approximation 311
10.8 A polynomial procedure REFINE 315
10.9 The minimum mean cycle cancelling algorithm 322
10.10 Some further problems 327
10.11 An application: Graphical codes 329

11 The Network Simplex Algorithm 343
11.1 The minimum cost flow problem 344
11.2 Tree solutions 346
11.3 Constructing an admissible tree structure 349
11.4 The algorithm 353
11.5 Efficient implementations 358

12 Synthesis of Networks 363
12.1 Symmetric networks 363
12.2 Synthesis of equivalent flow trees 366
12.3 Synthesizing minimal networks 373
12.4 Cut trees 379
12.5 Increasing the capacities 383

13 Matchings 387
13.1 The 1-factor theorem 387
13.2 Augmenting paths 390
13.3 Alternating trees and blossoms 394
13.4 The algorithm of Edmonds 400
13.5 Matching matroids 416

14 Weighted matchings 419
14.1 The bipartite case 420
14.2 The Hungarian algorithm 421
14.3 Matchings, linear programs, and polytopes 430
14.4 The general case 434
14.5 The Chinese postman 438
14.6 Matchings and shortest paths 442
14.7 Some further problems 449
14.8 An application: Decoding graphical codes 452

15 A Hard Problem: The TSP 457
15.1 Basic definitions 457
15.2 Lower bounds: Relaxations 460
15.3 Lower bounds: Subgradient optimization 466
15.4 Approximation algorithms 471
15.5 Upper bounds: Heuristics 477
15.6 Upper bounds: Local search 480
15.7 Exact neighborhoods and suboptimality 483
15.8 Optimal solutions: Branch and bound 489
15.9 Concluding remarks 497

A Some NP-Complete Problems 501

B Solutions 509
B.1 Solutions for Chapter 1 509
B.2 Solutions for Chapter 2 515
B.3 Solutions for Chapter 3 520
B.4 Solutions for Chapter 4 527
B.5 Solutions for Chapter 5 532
B.6 Solutions for Chapter 6 535
B.7 Solutions for Chapter 7 545
B.8 Solutions for Chapter 8 554
B.9 Solutions for Chapter 9 560
B.10 Solutions for Chapter 10 563
B.11 Solutions for Chapter 11 572
B.12 Solutions for Chapter 12 572
B.13 Solutions for Chapter 13 578
B.14 Solutions for Chapter 14 583

B.15 Solutions for Chapter 15 589

C List of Symbols 593
C.1 General Symbols 593
C.2 Special Symbols 595

References 601

Index 635

1

Basic Graph Theory

It is time to get back to basics.
John Major

Graph theory began in 1736 when Leonhard Euler (1707–1783) solved the well-known *Königsberg bridge problem* [Eul36][1]. This problem asked for a circular walk through the town of Königsberg (now Kaliningrad) in such a way as to cross over each of the seven bridges spanning the river Pregel once, and only once; see Figure 1.1 for a rough sketch of the situation.

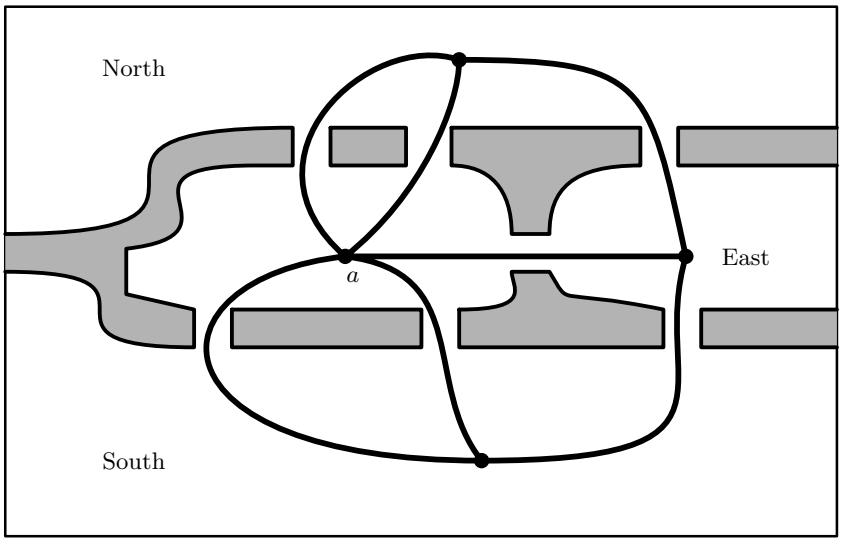


Fig. 1.1. The Königsberg bridge problem

When trying to solve this problem one soon gets the feeling that there is no solution. But how can this be proved? Euler realized that the precise shapes

[1] see [Wil86] and [BiLW76].

of the island and the other three territories involved are not important; the solvability depends only on their *connection properties*. Let us represent the four territories by points (called *vertices*), and the bridges by curves joining the respective points; then we get the *graph* also drawn in Figure 1.1. Trying to arrange a circular walk, we now begin a tour, say, at the vertex called a. When we return to a for the first time, we have used two of the five bridges connected with a. At our next return to a we have used four bridges. Now we can leave a again using the fifth bridge, but there is no possibility to return to a without using one of the five bridges a second time. This shows that the problem is indeed unsolvable. Using a similar argument, we see that it is also impossible to find *any* walk – not necessarily circular, so that the tour might end at a vertex different from where it began – which uses each bridge exactly once. Euler proved even more: he gave a necessary and sufficient condition for an arbitrary graph to admit a circular tour of the above kind. We will treat his theorem in Section 1.3. But first, we have to introduce some basic notions.

The present chapter contains a lot of definitions. We urge the reader to work on the exercises to get a better idea of what the terms really mean. Even though this chapter has an introductory nature, we will also prove a couple of nontrivial results and give two interesting applications. We warn the reader that the terminology in graph theory lacks universality, although this improved a little after the book by Harary [Har69] appeared.

1.1 Graphs, subgraphs and factors

A *graph* G is a pair $G = (V, E)$ consisting of a finite[2] set $V \neq \emptyset$ and a set E of two-element subsets of V. The elements of V are called *vertices*. An element $e = \{a, b\}$ of E is called an *edge* with *end vertices* a and b. We say that a and b are *incident* with e and that a and b are *adjacent* or *neighbors* of each other, and write $e = ab$ or $a \overset{e}{—} b$.

Let us mention two simple but important series of examples. The *complete graph* K_n has n vertices (that is, $|V| = n$) and all two-element subsets of V as edges. The *complete bipartite graph* $K_{m,n}$ has as vertex set the disjoint union of a set V_1 with m elements and a set V_2 with n elements; edges are all the sets $\{a, b\}$ with $a \in V_1$ and $b \in V_2$.

We will often illustrate graphs by pictures in the plane. The vertices of a graph $G = (V, E)$ are represented by (bold type) points and the edges by lines (preferably straight lines) connecting the end points. We give some examples in Figure 1.2. We emphasize that in these pictures the lines merely serve to indicate the vertices with which they are incident. In particular, the *inner points* of these lines as well as possible points of intersection of two edges (as in Figure 1.2 for the graphs K_5 and $K_{3,3}$) are not significant. In Section 1.5 we

[2] In graph theory, infinite graphs are studied as well. However, we restrict ourselves in this book – like [Har69] – to the finite case.

will study the question which graphs can be drawn without such additional points of intersection.

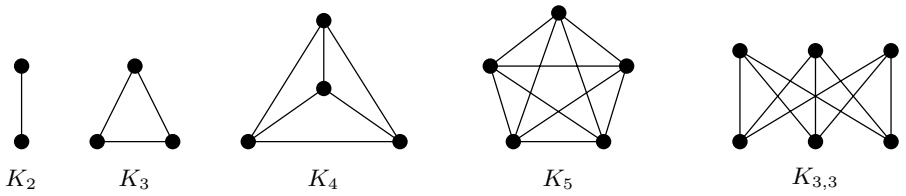


Fig. 1.2. Some graphs

Let $G = (V, E)$ be a graph and V' be a subset of V. By $E|V'$ we denote the set of all edges $e \in E$ which have both their vertices in V'. The graph $(V', E|V')$ is called the *induced subgraph* on V' and is denoted by $G|V'$. Each graph of the form (V', E') where $V' \subset V$ and $E' \subset E|V'$ is said to be a *subgraph* of G, and a subgraph with $V' = V$ is called a *spanning subgraph*. Some examples are given in Figure 1.3.

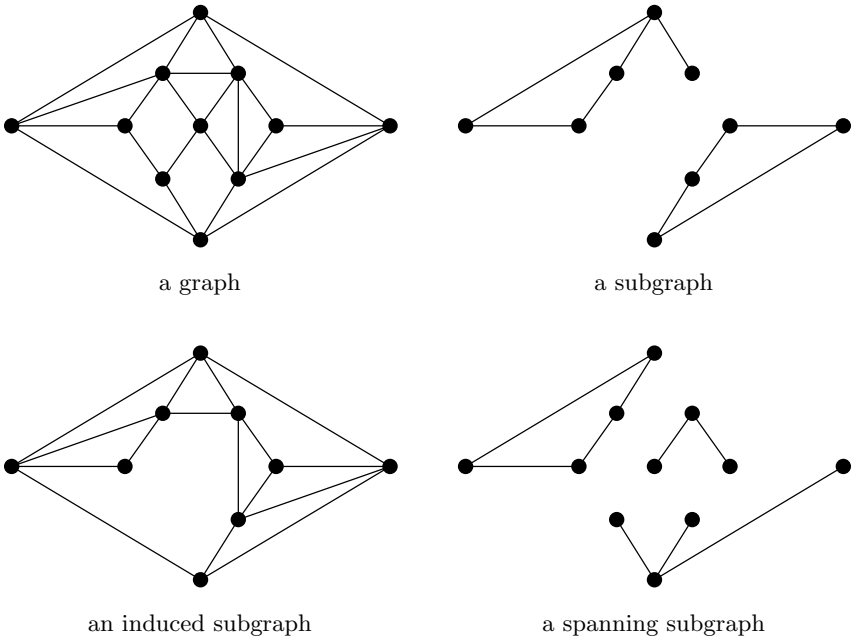


Fig. 1.3. Subgraphs

Given any vertex v of a graph, the *degree of* v, $\deg v$, is the number of edges incident with v. We can now state our first – albeit rather simple – result:

Lemma 1.1.1. *In any graph, the number of vertices of odd degree is even.*

Proof. Summing the degree over all vertices v, each edge is counted exactly twice, once for each of its vertices; thus $\sum_v \deg v = 2|E|$. As the right hand side is even, the number of odd terms $\deg v$ in the sum on the left hand side must also be even. □

If all vertices of a graph G have the same degree (say r), G is called a *regular* graph, more precisely an *r-regular* graph. The graph K_n is $(n-1)$-regular, the graph $K_{m,n}$ is regular only if $m = n$ (in which case it is n-regular). A *k-factor* is a k-regular spanning subgraph. If the edge set of a graph can be divided into k-factors, such a decomposition is called a *k-factorization* of the graph. A 1-factorization is also called a *factorization* or a *resolution*. Obviously, a 1-factor can exist only if G has an even number of vertices. Factorizations of K_{2n} may be interpreted as schedules for a tournament of $2n$ teams (in soccer, basketball etc.). The following exercise shows that such a factorization exists for all n. The problem of setting up schedules for tournaments will be studied in Section 1.7 as an application.

Exercise 1.1.2. We use $\{\infty, 1, \ldots, 2n-1\}$ as the vertex set of the complete graph K_{2n} and divide the edge set into subsets F_i for $i = 1, \ldots, 2n-1$, where $F_i = \{\infty i\} \cup \{jk : j + k \equiv 2i \pmod{2n-1}\}$. Show that the F_i form a factorization of K_{2n}. The case $n = 3$ is shown in Figure 1.4. Factorizations were first introduced by [Kir47]; interesting surveys are given by [MeRo85] and [Wal92].

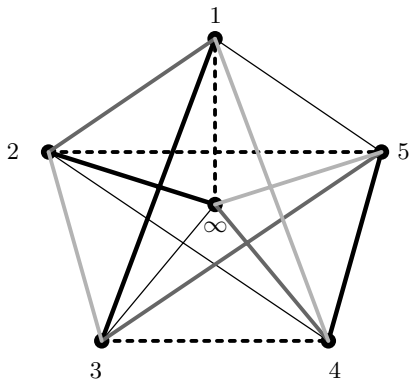


Fig. 1.4. A factorization of K_6

Let us conclude this section with two more exercises. First, we introduce a further family of graphs. The *triangular graph* T_n has as vertices the two-element subsets of a set with n elements. Two of these vertices are adjacent if and only if their intersection is not empty. Obviously, T_n is a $(2n-4)$-regular graph. But T_n has even stronger regularity properties: the number of vertices adjacent to two given vertices x, y depends only on whether x and y themselves are adjacent or not. Such a graph is called a *strongly regular graph*, abbreviated by *SRG*. These graphs are of great interest in finite geometry; see the books [CaLi91] and [BeJL99]. We will limit our look at SRG's in this book to a few exercises.

Exercise 1.1.3. Draw the graphs T_n for $n = 3, 4, 5$ and show that T_n has parameters $a = 2n-4$, $c = n-2$ and $d = 4$, where a is the degree of any vertex, c is the number of vertices adjacent to both x and y if x and y are adjacent, and d is the number of vertices adjacent to x and y if x and y are not adjacent.

For the next exercise, we need another definition. For a graph $G = (V, E)$, we will denote by $\binom{V}{2}$ the set of all pairs of its vertices. The graph $\overline{G} = (V, \binom{V}{2} \setminus E)$ is called the *complementary graph*. Two vertices of V are adjacent in $\overline{G}$ if and only if they are not adjacent in G.

Exercise 1.1.4. Let G be an SRG with parameters a, c, and d having n vertices. Show that $\overline{G}$ is also an SRG and determine its parameters. Moreover, prove the formula

$$a(a-c-1) = (n-a-1)d.$$

Hint: Count the number of edges yz for which y is adjacent to a given vertex x, whereas z is not adjacent to x.

1.2 Paths, cycles, connectedness, trees

Before we can go on to the theorem of Euler mentioned in Section 1.1, we have to formalize the idea of a circular tour. Let $(e_1, \ldots, e_n)$ be a sequence of edges in a graph G. If there are vertices $v_0, \ldots, v_n$ such that $e_i = v_{i-1}v_i$ for $i = 1, \ldots, n$, the sequence is called a *walk*; if $v_0 = v_n$, one speaks of a *closed walk*. A walk for which the e_i are distinct is called a *trail*, and a closed walk with distinct edges is a *closed trail*. If, in addition, the v_j are distinct, the trail is a *path*. A closed trail with $n \geq 3$, for which the v_j are distinct (except, of course, $v_0 = v_n$), is called a *cycle*. In any of these cases we use the notation

$$W\colon \quad v_0 \overset{e_1}{\text{——}} v_1 \overset{e_2}{\text{——}} v_2 \text{——} \ldots \text{——} v_{n-1} \overset{e_n}{\text{——}} v_n$$

and call n the *length* of W. The vertices v_0 and v_n are called the *start vertex* and the *end vertex* of W, respectively. We will sometimes specify a walk by

its sequence of vertices $(v_0, \ldots, v_n)$, provided that $v_{i-1}v_i$ is an edge for $i = 1, \ldots, n$. In the graph of Figure 1.5, (a, b, c, v, b, c) is a walk, but not a trail; and (a, b, c, v, b, u) is a trail, but not a path. Also, (a, b, c, v, b, u, a) is a closed trail, but not a cycle, whereas (a, b, c, w, v, u, a) is a cycle. The reader might want to consider some more examples.

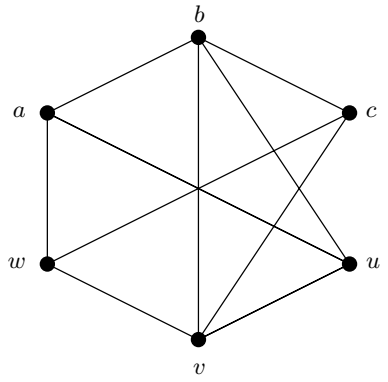


Fig. 1.5. An example for walks

Exercise 1.2.1. Show that any walk with start vertex a and end vertex b, where $a \neq b$, contains a path from a to b. Also prove that any closed walk of odd length contains a cycle. What do closed walks not containing a cycle look like?

Two vertices a and b of a graph G are called *connected* if there exists a walk with start vertex a and end vertex b. If all pairs of vertices of G are connected, G itself is called *connected*. For any vertex a, we consider (a) as a trivial walk of length 0, so that any vertex is connected with itself. Thus connectedness is an equivalence relation on the vertex set of G. The equivalence classes of this relation are called the *connected components* of G. Thus G is connected if and only if its vertex set V is its unique connected component. Components which contain only one vertex are also called *isolated vertices*. Let us give some exercises concerning these definitions.

Exercise 1.2.2. Let G be a graph with n vertices and assume that each vertex of G has degree at least $(n-1)/2$. Show that G must be connected.

Exercise 1.2.3. A graph G is connected if and only if there exists an edge $e = vw$ with $v \in V_1$ and $w \in V_2$ whenever $V = V_1 \dot{\cup} V_2$ (that is, $V_1 \cap V_2 = \emptyset$) is a decomposition of the vertex set of G.

Exercise 1.2.4. If G is not connected, the complementary graph $\overline{G}$ is connected.

If a and b are two vertices in the same connected component of a graph G, there has to exist a path of shortest length (say d) between a and b. (Why?) Then a and b are said to have *distance* $d = d(a, b)$. The notion of distances in a graph is fundamental; we will study it (and a generalization) thoroughly in Chapter 3.

In the remainder of this section, we will investigate the minimal connected graphs. First, some more definitions and an exercise. A graph is called *acyclic* if it does not contain a cycle. For a subset T of the vertex set V of a graph G we denote by $G \setminus T$ the induced subgraph on $V \setminus T$. This graph arises from G by omitting all vertices in T and all edges incident with these vertices. For a one-element set $T = \{v\}$ we write $G \setminus v$ instead of $G \setminus \{v\}$.

Exercise 1.2.5. Let G be a graph having n vertices, none of which are isolated, and $n-1$ edges, where $n \geq 2$. Show that G contains at least two vertices of degree 1.

Lemma 1.2.6. *A connected graph on n vertices has at least $n - 1$ edges.*

Proof. We use induction on n; the case $n = 1$ is trivial. Thus let G be a connected graph on $n \geq 2$ vertices. Choose an arbitrary vertex v of G and consider the graph $H = G \setminus v$. Note that H is not necessarily connected. Suppose H has connected components Z_i having n_i vertices ($i = 1, \ldots, k$), that is, $n_1 + \ldots + n_k = n - 1$. By induction hypothesis, the subgraph of H induced on Z_i has at least $n_i - 1$ edges. Moreover, v must be connected in G with each of the components Z_i by at least one edge. Thus G contains at least $(n_1 - 1) + \ldots + (n_k - 1) + k = n - 1$ edges. □

Lemma 1.2.7. *An acyclic graph on n vertices has at most $n - 1$ edges.*

Proof. If $n = 1$ or $E = \emptyset$, the statement is obvious. For the general case, choose any edge $e = ab$ in G. Then the graph $H = G \setminus e$ has exactly one more connected component than G. (Note that there cannot be a path in H from a to b, because such a path together with the edge e would give rise to a cycle in G.) Thus, H can be decomposed into connected, acyclic graphs $H_1, \ldots, H_k$ (where $k \geq 2$). By induction, we may assume that each graph H_i contains at most $n_i - 1$ edges, where n_i denotes the number of vertices of H_i. But then G has at most

$$(n_1 - 1) + \ldots + (n_k - 1) + 1 \;=\; (n_1 + \ldots + n_k) - (k - 1) \;\leq\; n - 1$$

edges. □

Theorem 1.2.8. *Let G be a graph with n vertices. Then any two of the following conditions imply the third:*

(a) *G is connected.*
(b) *G is acyclic.*
(c) *G has $n - 1$ edges.*

Proof. First let G be acyclic and connected. Then Lemmas 1.2.6 and 1.2.7 imply that G has exactly $n-1$ edges.

Next let G be a connected graph with $n-1$ edges. Suppose G contains a cycle C and consider the graph $H = G \setminus e$, where e is some edge of C. Then H is a connected with n vertices and $n-2$ edges, contradicting Lemma 1.2.6.

Finally, let G be an acyclic graph with $n-1$ edges. Then Lemma 1.2.7 implies that G cannot contain an isolated vertex, as omitting such a vertex would give an acyclic graph with $n-1$ vertices and $n-1$ edges. Now Exercise 1.2.5 shows that G has a vertex of degree 1, so that $G \setminus v$ is an acyclic graph with $n-1$ vertices and $n-2$ edges. By induction it follows that $G \setminus v$ and hence G are connected. □

Exercise 1.2.9. Give a different proof for Lemma 1.2.6 using the technique of omitting an edge e from G.

A graph T for which the conditions of Theorem 1.2.8 hold is called a *tree*. A vertex of T with degree 1 is called a *leaf*. A *forest* is a graph whose connected components are trees. We will have a closer look at trees in Chapter 4.

In Section 4.2 we will use rather sophisticated techniques from linear algebra to prove a formula for the number of trees on n vertices; this result is usually attributed to Cayley [Cay89], even though it is essentially due to Borchardt [Bor60]. Here we will use a more elementary method to prove a stronger result – which is indeed due to Cayley. By $f(n,s)$ we denote the number of forests G having n vertices and exactly s connected components, for which s fixed vertices are in distinct components; in particular, the number of trees on n vertices is $f(n,1)$. Cayley's theorem gives a formula for the numbers $f(n,s)$; we use a simple proof taken from [Tak90a].

Theorem 1.2.10. *One has* $f(n,s) = sn^{n-s-1}$.

Proof. We begin by proving the following recursion formula:

$$f(n,s) \;=\; \sum_{j=0}^{n-s} \binom{n-s}{j} f(n-1, s+j-1), \tag{1.1}$$

where we put $f(1,1) = 1$ and $f(n,0) = 0$ for $n \geq 1$. How can an arbitrary forest G with vertex set $V = \{1,\ldots,n\}$ having precisely s connected components be constructed? Let us assume that the vertices $1,\ldots,s$ are the specified vertices which belong to distinct components. The degree of vertex 1 can take the values $j = 0,\ldots,n-s$, as the neighbors of 1 may form an arbitrary subset $\Gamma(1)$ of $\{s+1,\ldots,n\}$. Then we have – after choosing the degree j of 1 – exactly $\binom{n-s}{j}$ possibilities to choose $\Gamma(1)$. Note that the graph $G \setminus 1$ is a forest with vertex set $V \setminus \{1\} = \{2,\ldots,n\}$ and exactly $s+j-1$ connected components, where the vertices $2,\ldots s$ and the j elements of $\Gamma(1)$ are in different connected components. After having chosen j and $\Gamma(1)$, we still have

$f(n-1, s+j-1)$ possibilities to construct the forest $G \setminus 1$. This proves the recursion formula (1.1).

We now prove the desired formula for the $f(n,s)$ by using induction on n. The case $n = 1$ is trivial. Thus we let $n \geq 2$ and assume that

$$f(n-1,i) \;=\; i(n-1)^{n-i-2} \qquad \text{holds for} \quad i = 1, \ldots n-1. \tag{1.2}$$

Using this in equation (1.1) gives

$$\begin{aligned}
f(n,s) \;&=\; \sum_{j=0}^{n-s} \binom{n-s}{j} (s+j-1)(n-1)^{n-s-j-1} \\
&=\; \sum_{j=1}^{n-s} j \binom{n-s}{j} (n-1)^{n-s-j-1} \\
&\qquad\qquad + (s-1) \sum_{j=0}^{n-s} \binom{n-s}{j} (n-1)^{n-s-j-1} \\
&=\; (n-s) \sum_{j=1}^{n-s} \binom{n-s-1}{j-1} (n-1)^{n-s-j-1} \\
&\qquad\qquad + (s-1) \sum_{j=0}^{n-s} \binom{n-s}{j} (n-1)^{n-s-j-1} \\
&=\; \frac{n-s}{n-1} \sum_{k=0}^{n-s-1} \binom{n-s-1}{k} (n-1)^{(n-s-1)-k} \times 1^k \\
&\qquad\qquad + \frac{s-1}{n-1} \sum_{j=0}^{n-s} \binom{n-s}{j} (n-1)^{n-s-j} \times 1^j \\
&=\; \frac{(n-s)n^{n-s-1} + (s-1)n^{n-s}}{n-1} \;=\; sn^{n-s-1}.
\end{aligned}$$

This proves the theorem. □

Note that the rather tedious calculations in the induction step may be replaced by the following – not shorter, but more elegant – combinatorial argument. We have to split up the sum we got from using equation (1.2) in (1.1) in a different way:

$$\begin{aligned}
f(n,s) \;&=\; \sum_{j=0}^{n-s} \binom{n-s}{j} (s+j-1)(n-1)^{n-s-j-1} \\
&=\; \sum_{j=0}^{n-s} \binom{n-s}{j} (n-1)^{n-s-j} \\
&\qquad\qquad - \sum_{j=0}^{n-s-1} \binom{n-s}{j} (n-s-j)(n-1)^{n-s-j-1}.
\end{aligned}$$

Now the first sum counts the number of words of length $n-s$ over the alphabet $V = \{1, \dots, n\}$, as the binomial coefficient counts the number of possibilities for distributing j entries 1 (where j has to be between 0 and $n-s$), and the factor $(n-1)^{n-s-j}$ gives the number of possibilities for filling the remaining $n-s-j$ positions with entries $\neq 1$. Similarly, the second sum counts the number of words of length $n-s$ over the alphabet $V = \{0, 1, \dots, n\}$ which contain exactly one entry 0. As there are obvious formulas for these numbers, we directly get

$$f(n,s) \;=\; n^{n-s} - (n-s)n^{n-s-1} \;=\; sn^{n-s-1}.$$

Borchardt's result is now an immediate consequence of Theorem 1.2.10:

Corollary 1.2.11. *The number of trees on n vertices is n^{n-2}.* □

It is interesting to note that n^{n-2} is also the cardinality of the set $\mathbf{W}$ of words of length $n-2$ over an alphabet V with n elements, which suggests that we might prove Corollary 1.2.11 by constructing a bijection between $\mathbf{W}$ and the set $\mathbf{T}$ of trees with vertex set V. This is indeed possible as shown by Prüfer [Pru18]; we will follow the account in [Lue89] and construct the *Prüfer code* $\pi_V \colon \mathbf{T} \to \mathbf{W}$ recursively. As we will need an ordering of the elements of V, we assume in what follows, without loss of generality, that V is a subset of $\mathbb{N}$.

Thus let $G = (V, E)$ be a tree. For $n = 2$ the only tree on V is mapped to the empty word; that is, we put $\pi_V(G) = ()$. For $n \geq 3$ we use the smallest leaf of G to construct a tree on $n-1$ vertices. We write

$$v \;=\; v(G) \;=\; \min\{u \in V \colon \deg_G(u) = 1\} \tag{1.3}$$

and denote by $e = e(G)$ the unique edge incident with v, and by $w = w(G)$ the other end vertex of e. Now let $G' = G \setminus v$. Then G' has $n-1$ vertices, and we may assume by induction that we know the word corresponding to G' under the Prüfer code on $V' = V \setminus \{v\}$. Hence we can define recursively

$$\pi_V(G) \;=\; (w, \pi_{V'}(G')). \tag{1.4}$$

It remains to show that we have indeed constructed the desired bijection. We need the following lemma which allows us to determine the minimal leaf of a tree G on V from its Prüfer code.

Lemma 1.2.12. *Let G be a tree on V. Then the leaves of G are precisely those elements of V which do not occur in $\pi_V(G)$. In particular,*

$$v(G) \;=\; \min\{u \in V \colon u \textit{ does not occur in } \pi_V(G)\}. \tag{1.5}$$

Proof. First suppose that an element u of V occurs in $\pi_V(G)$. Then u was added to $\pi_V(G)$ at some stage of our construction; that is, some subtree H of G was considered, and u was adjacent to the minimal leaf $v(H)$ of H. Now

if u were also a leaf of G (and thus of H), then H would have to consist only of u and $v(G)$, so that H would have the empty word as Prüfer code, and u would not occur in $\pi_V(G)$, contradicting our assumption.

Now suppose that u is not a leaf. Then there is at least one edge incident with u which is discarded during the construction of the Prüfer code of G, since the construction only ends when a tree on two vertices remains of G. Let e be the edge incident with u which is omitted first. At that point of the construction, u is not a leaf, so that the other end vertex of e has to be the minimal leaf of the respective subtree. But then, by our construction, u is used as the next coordinate in $\pi_V(G)$. □

Theorem 1.2.13. *The Prüfer code $\pi_V : \mathbf{T} \to \mathbf{W}$ defined by equations (1.3) and (1.4) is a bijection.*

Proof. For $n = 2$, the statement is clear, so let $n \geq 3$. First we show that π_V is surjective. Let $\mathbf{w} = (w_1, \ldots, w_{n-2})$ be an arbitrary word over V, and denote by v the smallest element of V which does not occur as a coordinate in $\mathbf{w}$. By induction, we may assume that there is a tree G' on the vertex set $V' = V \setminus \{v\}$ with $\pi_{V'}(G') = (w_2, \ldots, w_{n-2})$. Now we add the edge $e = vw_1$ to G' (as Lemma 1.2.12 suggests) and get a tree G on V. It is easy to verify that $v = v(G)$ and thus $\pi_V(G) = \mathbf{w}$. To prove injectivity, let G and H be two trees on $\{1, \ldots, n\}$ and suppose $\pi_V(G) = \pi_V(H)$. Now let v be the smallest element of V which does not occur in $\pi_V(G)$. Then Lemma 1.2.12 implies that $v = v(G) = v(H)$. Thus G and H both contain the edge $e = vw$, where w is the first entry of $\pi_V(G)$. Then G' and H' are both trees on $V' = V \setminus \{v\}$, and we have $\pi_{V'}(G') = \pi_{V'}(H')$. Using induction, we conclude $G' = H'$ and hence $G = H$. □

Note that the proof of Theorem 1.2.13 together with Lemma 1.2.12 gives a constructive method for decoding the Prüfer code.

Example 1.2.14. Figure 1.6 shows some trees and their Prüfer codes for $n = 6$ (one for each isomorphism class, see Exercise 4.1.6).

Exercise 1.2.15. Determine the trees with vertex set $\{1, \ldots, n\}$ corresponding to the following Prüfer codes: $(1, 1, \ldots, 1)$; $(2, 3, \ldots, n-2, n-1)$; $(2, 3, \ldots, n-3, n-2, n-2)$; $(3, 3, 4, \ldots, n-3, n-2, n-2)$.

Exercise 1.2.16. How can we determine the degree of an arbitrary vertex u of a tree G from its Prüfer code $\pi_V(G)$? Give a condition for $\pi_V(G)$ to correspond to a path or a star (where a *star* is a tree having one exceptional vertex z which is adjacent to all other vertices).

Exercise 1.2.17. Let $(d_1, \ldots, d_n)$ be a sequence of positive integers. Show that there is a tree on n vertices having degrees $d_1, \ldots, d_n$ if and only if

$$d_1 + \ldots + d_n \;=\; 2(n-1), \tag{1.6}$$

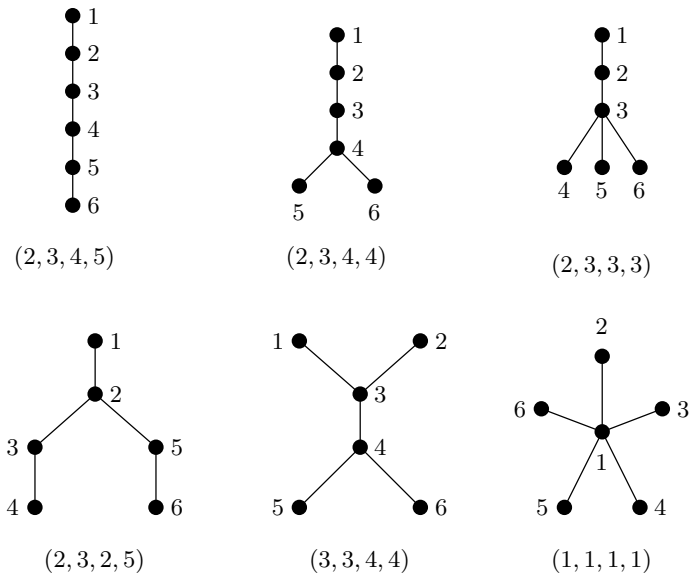


Fig. 1.6. Some trees and their Prüfer codes

and construct a tree with degree sequence $(1, 1, 1, 1, 2, 3, 3)$. Hint: Use the Prüfer code.

We remark that the determination of the possible degree sequences for arbitrary graphs on n vertices is a considerably more difficult problem; see, for instance, [SiHo91] and [BaSa95].

We have now seen two quite different proofs for Corollary 1.2.11 which illustrate two important techniques for solving enumeration problems, namely using recursion formulas on the one hand and using bijections on the other. In Section 4.2 we will see yet another proof which will be based on the application of algebraic tools (like matrices and determinants). In this text, we cannot treat the most important tool of enumeration theory, namely generating functions. The interested reader can find the basics of enumeration theory in any good book on combinatorics; for a more thorough study we recommend the books by Stanley [Sta86, Sta99] or the extensive monograph [GoJa83], all of which are standard references.

Let us also note that the number $f(n)$ of forests on n vertices has been studied several times; see [Tak90b] and the references given there. Takács proves the following simple formula which is, however, not at all easy to derive:

$$f(n) \;=\; \frac{n!}{n+1} \sum_{j=0}^{\lfloor n/2 \rfloor} (-1)^j \frac{(2j+1)(n+1)^{n-2j}}{2^j j!(n-2j)!}.$$

Finally, me mention an interesting asymptotic result due to Rényi [Ren59] which compares the number of all forests with the number of all trees:

$$\lim_{n\to\infty} \frac{f(n)}{n^{n-2}} = \sqrt{e} \approx 1.6487.$$

1.3 Euler tours

In this section we will solve the Königsberg bridge problem for arbitrary graphs. The reader should note that Figure 1.1 does not really depict a graph according to the definitions given in Section 1.1, because there are pairs of vertices which are connected by more than one edge. Thus we generalize our definition as follows. Intuitively, for a *multigraph* on a vertex set V, we want to replace the edge set of an ordinary graph by a family E of two-element subsets of V. To be able to distinguish different edges connecting the same pair of vertices, we formally define a *multigraph* as a triple (V, E, J), where V and E are disjoint sets, and J is a mapping from E to the set of two-element subsets of V, the *incidence map*. The image $J(e)$ of an edge e is the set $\{a, b\}$ of end vertices of e. Edges e and e' with $J(e) = J(e')$ are called *parallel*. Then all the notions introduced so far carry over to multigraphs. However, in this book we will – with just a few exceptions – restrict ourselves to graphs.[3]

The circular tours occurring in the Königsberg bridge problem can be described abstractly as follows. An *Eulerian trail* of a multigraph G is a trail which contains each edge of G (exactly once, of course); if the trail is closed, then it is called an *Euler tour*.[4] A multigraph is called *Eulerian* if it contains an Euler tour. The following theorem of [Eul36] characterizes the Eulerian multigraphs.

Theorem 1.3.1 (Euler's theorem). *Let G be a connected multigraph. Then the following statements are equivalent:*

(a) *G is Eulerian.*
(b) *Each vertex of G has even degree.*
(c) *The edge set of G can be partitioned into cycles.*

Proof: We first assume that G is Eulerian and pick an Euler tour, say C. Each occurrence of a vertex v in C adds 2 to its degree. As each edge of G occurs exactly once in C, all vertices must have even degree. The reader should work out this argument in detail.

[3] Some authors denote the structure we call a *multigraph* by *graph*; graphs according to our definition are then called *simple graphs*. Moreover, sometimes even edges e for which $J(e)$ is a set $\{a\}$ having only one element are admitted; such edges are then called *loops*. The corresponding generalization of multigraphs is often called a *pseudograph*.

[4] Sometimes one also uses the term *Eulerian cycle*, even though an Euler tour usually contains vertices more than once.

Next suppose that (b) holds and that G has n vertices. As G is connected, it has at least $n-1$ edges by Lemma 1.2.6. Since G does not contain vertices of degree 1, it actually has at least n edges, by Exercise 1.2.5. Then Lemma 1.2.7 shows that there is a cycle K in G. Removing K from G we get a graph H in which all vertices again have even degree. Considering the connected components of H separately, we may – using induction – partition the edge set of H into cycles. Hence, the edge set of G can be partitioned into cycles.

Finally, assume the validity of (c) and let C be one of the cycles in the partition of the edge set E into cycles. If C is an Euler tour, we are finished. Otherwise there exists another cycle C' having a vertex v in common with C. We can w.l.o.g. use v as start and end vertex of both cycles, so that CC' (that is, C followed by C') is a closed trail. Continuing in the same manner, we finally reach an Euler tour. □

Corollary 1.3.2. *Let G be a connected multigraph with exactly $2k$ vertices of odd degree. Then G contains an Eulerian trail if and only if $k = 0$ or $k = 1$.*

Proof: The case $k = 0$ is clear by Theorem 1.3.1. So suppose $k \neq 0$. Similar to the proof of Theorem 1.3.1 it can be shown that an Eulerian trail can exist only if $k = 1$; in this case the Eulerian trail has the two vertices of odd degree as start and end vertices. Let $k = 1$ and name the two vertices of odd degree a and b. By adding an (additional) edge ab to G, we get a connected multigraph H whose vertices all have even degree. Hence H contains an Euler tour C by Theorem 1.3.1. Omitting the edge ab from C then gives the desired Eulerian trail in G. □

Exercise 1.3.3. Let G be a connected multigraph having exactly $2k$ vertices of odd degree ($k \neq 0$). Then the edge set of G can be partitioned into k trails.

The *line graph* $L(G)$ of a graph G has as vertices the edges of G; two edges of G are adjacent in $L(G)$ if and only if they have a common vertex in G. For example, the line graph of the complete graph K_n is the triangular graph T_n.

Exercise 1.3.4. Give a formula for the degree of a vertex of $L(G)$ (using the degrees in G). In which cases is $L(K_{m,n})$ an SRG?

Exercise 1.3.5. Let G be a connected graph. Find a necessary and sufficient condition for $L(G)$ to be Eulerian. Conclude that the line graph of an Eulerian graph is likewise Eulerian, and show that the converse is false in general.

Finally we recommend the very nice survey [Fle83] which treats Eulerian graphs and a lot of related questions in detail; for another survey, see [LeOe86]. A much more extensive treatment of these subjects can be found in two monographs by Fleischner [Fle90, Fle91]. For a survey of line graphs, see [Pri96].

1.4 Hamiltonian cycles

In 1857 Sir William Rowan Hamilton (1805–1865, known to every mathematician for the quaternions and the theorem of Cayley–Hamilton) invented the following *Icosian game* which he then sold to a London game dealer in 1859 for 25 pounds; it was realized physically as a pegboard with holes. The corners of a regular dodecahedron are labelled with the names of cities; the task is to find a circular tour along the edges of the dodecahedron visiting each city exactly once, where sometimes the first steps of the tour might also be prescribed. More about this game can be found in [BaCo87]. We may model the Icosian game by looking for a cycle in the corresponding *dodecahedral graph* which contains each vertex exactly once. Such a cycle is therefore called a *Hamiltonian cycle*. In Figure 1.7 we give a solution for Hamilton's original problem.

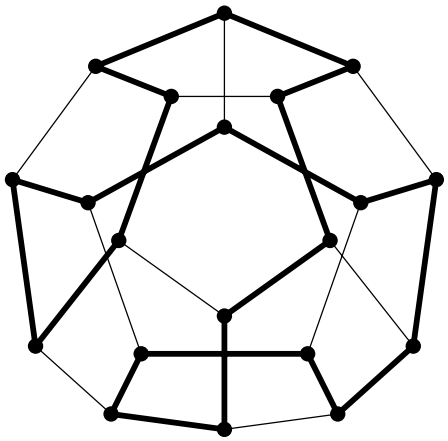


Fig. 1.7. The Icosian game

Although Euler tours and Hamiltonian cycles have similar definitions, they are quite different. For example, there is no nice characterization of *Hamiltonian graphs*; that is, of those graphs containing a Hamiltonian cycle. As we will see in the next chapter, there are good reasons to believe that such a good characterization cannot exist. However, we know many sufficient conditions for the existence of a Hamiltonian cycle; most of these conditions are statements about the degrees of the vertices. Obviously, the complete graph K_n is Hamiltonian.

We first prove a theorem from which we can derive several sufficient conditions on the sequence of degrees in a graph. Let G be a graph on n vertices. If G contains non-adjacent vertices u and v such that $\deg u + \deg v \geq n$, we add the edge uv to G. We continue this procedure until we get a graph

$[G]$, in which, for any two non-adjacent vertices x and y, we always have $\deg x + \deg y < n$. The graph $[G]$ is called the *closure* of G. (We leave it to the reader to show that $[G]$ is uniquely determined.) Then we have the following theorem due to Bondy and Chvátal [BoCh76].

Theorem 1.4.1. *A graph G is Hamiltonian if and only if its closure $[G]$ is Hamiltonian.*

Proof. If G is Hamiltonian, $[G]$ is obviously Hamiltonian. As $[G]$ is derived from G by adding edges sequentially, it will suffice to show that adding just one edge – as described above – does not change the fact whether a graph is Hamiltonian or not. Thus let u and v be two non-adjacent vertices with $\deg u + \deg v \geq n$, and let H be the graph which results from adding the edge uv to G. Suppose that H is Hamiltonian, but G is not. Then there exists a Hamiltonian cycle in H containing the edge uv, so that there is a path $(x_1, x_2, \ldots, x_n)$ in G with $x_1 = u$ and $x_n = v$ containing each vertex of G exactly once. Consider the sets

$$X = \{x_i : vx_{i-1} \in E \text{ and } 3 \leq i \leq n-1\}$$

and

$$Y = \{x_i : ux_i \in E \text{ and } 3 \leq i \leq n-1\}.$$

As u and v are not adjacent in G, we have $|X| + |Y| = \deg u + \deg v - 2 \geq n-2$. Hence there exists an index i with $3 \leq i \leq n-1$ such that vx_{i-1} as well as ux_i are edges in G. But then $(x_1, x_2, \ldots, x_{i-1}, x_n, x_{n-1}, \ldots, x_i, x_1)$ is a Hamiltonian cycle in G (see Figure 1.8), a contradiction. □

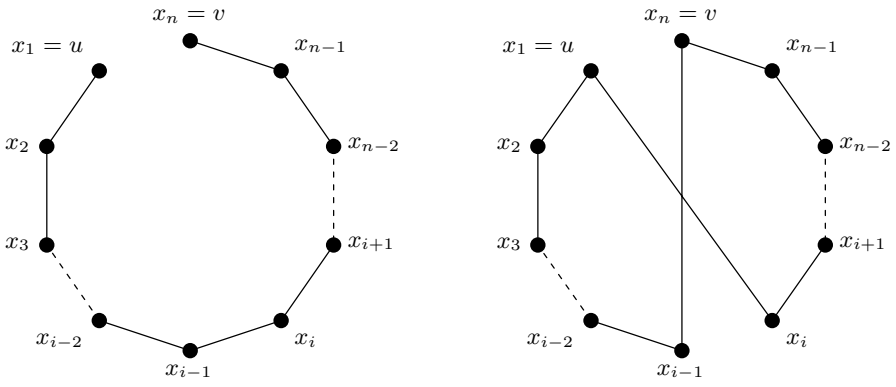


Fig. 1.8. Proof of Theorem 1.4.1

In general, it will not be much easier to decide whether $[G]$ is Hamiltonian. But if, for example, $[G]$ is a complete graph, G has to be Hamiltonian by

Theorem 1.4.1. Using this observation, we obtain the following two sufficient conditions for the existence of a Hamiltonian cycle due to Ore and Dirac [Ore60, Dir52], respectively.

Corollary 1.4.2. *Let G be a graph with $n \geq 3$ vertices. If $\deg u + \deg v \geq n$ holds for any two non-adjacent vertices u and v, then G is Hamiltonian.* □

Corollary 1.4.3. *Let G be a graph with $n \geq 3$ vertices. If each vertex of G has degree at least $n/2$, then G is Hamiltonian.* □

Bondy and Chvátal used their Theorem 1.4.1 to derive further sufficient conditions for the existence of a Hamiltonian cycle; in particular, they obtained the earlier result of Las Vergnas [Las72] in this way. We also refer the reader to [Har69, Ber73, Ber78, GoMi84, Chv85] for more results about Hamiltonian graphs.

Exercise 1.4.4. Let G be a graph with n vertices and m edges, and assume $m \geq \frac{1}{2}(n-1)(n-2)+2$. Use Corollary 1.4.2 to show that G is Hamiltonian.

Exercise 1.4.5. Determine the minimal number of edges a graph G with six vertices must have if $[G]$ is the complete graph K_6.

Exercise 1.4.6. If G is Eulerian, then $L(G)$ is Hamiltonian. Does the converse hold?

We now digress a little and look at one of the oldest problems in recreational mathematics, the *knight's problem*. This problem consists of moving a knight on a chessboard – beginning, say, in the upper left corner – such that it reaches each square of the board exactly once and returns with its last move to the square where it started.[5] As mathematicians tend to generalize everything, they want to solve this problem for chess boards of arbitrary size, not even necessarily square. Thus we look at boards having $m \times n$ squares. If we represent the squares of the chessboard by vertices of a graph G and connect two squares if the knight can move directly from one of them to the other, a solution of the knight's problem corresponds to a Hamiltonian cycle in G. Formally, we may define G as follows. The vertices of G are the pairs (i,j) with $1 \leq i \leq m$ and $1 \leq j \leq n$; as edges we have all sets $\{(i,j),(i',j')\}$ with $|i-i'|=1$ and $|j-j'|=2$ or $|i-i'|=2$ and $|j-j'|=1$. Most of the vertices of G have degree 8, except the ones which are too close to the border of the chess-board. For example, the vertices at the corners have degree 2. In our context of Hamiltonian graphs, this interpretation of the knight's

[5] It seems that the first known knight's tours go back more than a thousand years to the Islamic and Indian world around 840–900. The first examples in the modern European literature occur in 1725 in Ozanam's book [Oza25], and the first mathematical analysis of knight's tours appears in a paper presented by Euler to the Academy of Sciences at Berlin in 1759 [Eul66]. See the excellent website by Jelliss [Jel03]; and [Wil89], an interesting account of the history of Hamiltonian graphs.

problem is of obvious interest. However, solving the problem is just as well possible without looking at it as a graph theory problem. Figure 1.9 gives a solution for the ordinary chess-board of $8 \times 8 = 64$ squares; the knight moves from square to square according to the numbers with which the squares are labelled. Figure 1.9 also shows the Hamiltonian cycle in the corresponding graph.

1	52	25	38	3	54	15	40
24	37	2	53	26	39	4	55
51	64	27	12	29	14	41	16
36	23	62	45	60	43	56	5
63	50	11	28	13	30	17	42
22	35	46	61	44	59	6	57
49	10	33	20	47	8	31	18
34	21	48	9	32	19	58	7

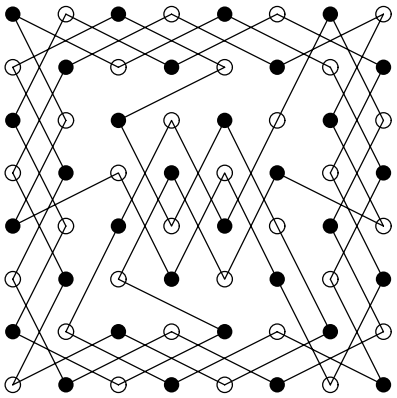


Fig. 1.9. A knight's cycle

The following theorem of Schwenk [Schw91] solves the knight's problem for arbitrary rectangular chessboards.

Result 1.4.7. *Every chessboard of size $m \times n$ (where $m \leq n$) admits a knight's cycle, with the following three exceptions:*

(a) *m and n are both odd;*

(b) *$m = 1, 2$ or 4;*

(c) *$m = 3$ and $n = 4, 6$ or 8.* □

The proof (which is elementary) is a nice example of how such problems can be solved recursively, combining the solutions for some small sized chessboards. Solutions for boards of sizes 3×10, 3×12, 5×6, 5×8, 6×6, 6×8, 7×6, 7×8 and 8×8 are needed, and these can easily be found by computer. The version of the knight's problem where no last move closing the cycle is required has also been studied; see [CoHMW92, CoHMW94].

Exercise 1.4.8. Show that knight's cycles are impossible for the cases (a) and (b) in Theorem 1.4.7. (Case (c) is more difficult.) Hint: For case (a) use the ordinary coloring of a chessboard with black and white squares; for (b) use the same coloring as well as another appropriate coloring (say, in red and green squares) and look at a hypothetical knight's cycle.

We close this section with a first look at one of the most fundamental problems in combinatorial optimization, the *travelling salesman problem* (for short, the *TSP*). This problem will later serve as our standard example of a *hard* problem, whereas most of the other problems we will consider are *easy*.[6]

Imagine a travelling salesman who has to take a circular journey visiting n cities and wants to be back in his home city at the end of the journey. Which route is – knowing the distances between the cities – the best one? To translate this problem into the language of graph theory, we consider the cities as the vertices of the complete graph K_n; any circular tour then corresponds to a Hamiltonian cycle in K_n. To have a measure for the expense of a route, we give each edge e a *weight* $w(e)$. (This weight might be the distance between the cities, but also the time the journey takes, or the cost, depending on the criterion subject to which we want to optimize the route.) The expense of a route then is the sum of the weights of all edges in the corresponding Hamiltonian cycle. Thus our problem may be stated formally as follows.

Problem 1.4.9 (travelling salesman problem, TSP). Consider the complete graph K_n together with a weight function $w\colon E \to \mathbb{R}^+$. Find a cyclic permutation $(1, \pi(1), \dots, \pi^{n-1}(1))$ of the vertex set $\{1, \dots, n\}$ such that

$$w(\pi) := \sum_{i=1}^{n} w(\{i, \pi(i)\})$$

is minimal. We call any cyclic permutation π of $\{1, \dots, n\}$ as well as the corresponding Hamiltonian cycle

$$1 \quad — \quad \pi(1) \quad — \quad \dots \quad — \quad \pi^{n-1}(1) \quad — \quad 1$$

in K_n a *tour*. An *optimal tour* is a tour π such that $w(\pi)$ is minimal among all tours.

Note that looking at all the possibilities for tours would be a lot of work: even for only nine cities we have $8!/2 = 20160$ possibilities. (We can always take the tour to begin at vertex 1, and fix the direction of the tour.) Of course it would be feasible to examine all these tours – at least by computer. But for 20 cities, we already get about 10^{17} possible tours, making this brute force approach more or less impossible.

It is convenient to view Problem 1.4.9 as a problem concerning matrices, by writing the weights as a matrix $W = (w_{ij})$. Of course, we have $w_{ij} = w_{ji}$ and $w_{ii} = 0$ for $i = 1, \dots, n$. The instances of a TSP on n vertices thus correspond to the symmetric matrices in $(\mathbb{R}^+)^{(n,n)}$ with entries 0 on the main diagonal. In the following example we have rounded the distances between the nine cities Aachen, Basel, Berlin, Dusseldorf, Frankfurt, Hamburg, Munich, Nuremberg and Stuttgart to units of 10 kilometers; we write $10w_{ij}$ for the rounded distance.

[6] The distinction between *easy* and *hard* problems can be made quite precise; we will explain this in Chapter 2.

Example 1.4.10. Determine an optimal tour for

	Aa	Ba	Be	Du	Fr	Ha	Mu	Nu	St
Aa	0	57	64	8	26	49	64	47	46
Ba	57	0	88	54	34	83	37	43	27
Be	64	88	0	57	56	29	60	44	63
Du	8	54	57	0	23	43	63	44	41
Fr	26	34	56	23	0	50	40	22	20
Ha	49	83	29	43	50	0	80	63	70
Mu	64	37	60	63	40	80	0	17	22
Nu	47	43	44	44	22	63	17	0	19
St	46	27	63	41	20	70	22	19	0

An optimal tour and a tour which is slightly worse (obtained by replacing the edges $MuSt$ and $BaFr$ by the edges $MuBa$ and $StFr$) are shown in Figure 1.10. We will study the TSP in Chapter 15 in detail, always illustrating the various techniques which we encounter using the present example.

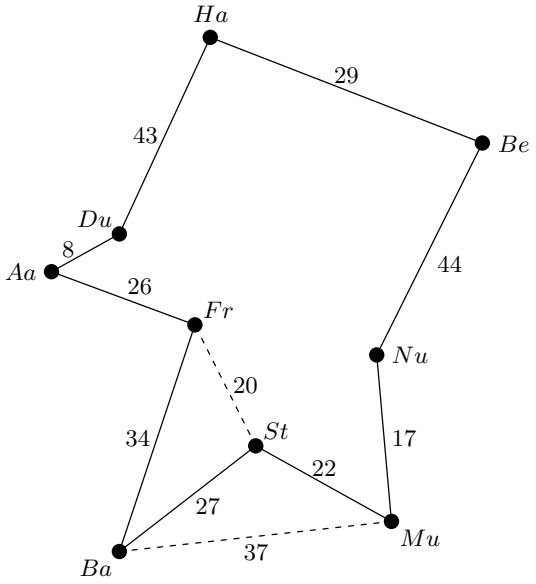


Fig. 1.10. Two tours for the TSP on 9 cities

Even though the number of possible tours grows exponentially with n, there still might be an easy method to solve the TSP. For example, the number of closed trails in a graph may also grow very fast as the number of edges

increases; but, as we will see in Chapter 2, it is still easy to find an Euler tour or to decide that no such tour exists. On the other hand, it is difficult to find Hamiltonian cycles. We will return to these examples in the next chapter to think about the complexity (that is, the degree of difficulty) of a problem.

1.5 Planar graphs

This section is devoted to the problem of drawing graphs in the plane. First, we need the notion of isomorphism. Two graphs $G = (V, E)$ and $G' = (V', E')$ are called *isomorphic* if there is a bijection $\alpha : V \to V'$ such that we have $\{a, b\} \in E$ if and only if $\{\alpha(a), \alpha(b)\} \in E'$ for all a, b in V. Let E be a set of line segments in three-dimensional Euclidean space and V the set of end points of the line segments in E. Identifying each line segment with the two-element set of its end points, we can consider (V, E) as a graph. Such a graph is called *geometric* if any two line segments in E are disjoint or have one of their end points in common.

Lemma 1.5.1. *Every graph is isomorphic to a geometric graph.*

Proof. Let $G = (V, E)$ be a graph on n vertices. Choose a set V' of n points in $\mathbb{R}^3$ such that no four points lie in a common plane (Why is that possible?) and map V bijectively to V'. Let E' contain, for each edge e in E, the line segment connecting the images of the vertices on e. It is easy to see that (V', E') is a geometric graph isomorphic to G. □

As we have only a plane piece of paper to draw graphs, Lemma 1.5.1 does not help us a lot. We call a geometric graph *plane* if its line segments all lie in one plane. Any graph isomorphic to a plane graph is called *planar*.[7] Thus, the planar graphs are exactly those graphs which can be drawn in the plane without additional points of intersection between the edges; see the comments after Figure 1.2. We will see that most graphs are not planar; more precisely, we will show that planar graphs can only contain comparatively few edges (compared to the number of vertices).

Let $G = (V, E)$ be a planar graph. If we omit the line segments of G from the plane surface on which G is drawn, the remainder splits into a number of connected open regions; the closure of such a region is called a *face*. The following theorem gives another famous result due to Euler [Eul52/53].

Theorem 1.5.2 (Euler's formula). *Let G be a connected planar graph with n vertices, m edges and f faces. Then $n - m + f = 2$.*

[7] In the definition of planar graphs, one often allows not only line segments, but curves as well. However, this does not change the definition of planarity as given above, see [Wag36]. For multigraphs, it is necessary to allow curves.

Proof. We use induction on m. For $m = 0$ we have $n = 1$ and $f = 1$, so that the statement holds. Now let $m \neq 0$. If G contains a cycle, we discard one of the edges contained in this cycle and get a graph G' with $n' = n$, $m' = m - 1$ and $f' = f - 1$. By induction hypothesis, $n' - m' + f' = 2$ and hence $n - m + f = 2$. If G is acyclic, then G is a tree so that $m = n - 1$, by Theorem 1.2.8; as $f = 1$, we again obtain $n - m + f = 2$. □

Originally, Euler's formula was applied to the vertices, edges and faces of a convex polyhedron; it is used, for example, to determine the five regular polyhedra (or *Platonic solids*, namely the tetrahedron, octahedron, cube, icosahedron and dodecahedron); see, for instance, [Cox73]. We will now use Theorem 1.5.2 to derive bounds on the number of edges of planar graphs. We need two more definitions. An edge e of a connected graph G is called a *bridge* if $G \setminus e$ is not connected. The *girth* of a graph containing cycles is the length of a shortest cycle.

Theorem 1.5.3. *Let G be a connected planar graph on n vertices. If G is acyclic, then G has precisely $n-1$ edges. If G has girth at least g, then G can have at most $\frac{g(n-2)}{g-2}$ edges.*

Proof. The first claim holds by Theorem 1.2.8. Thus let G be a connected planar graph having n vertices, m edges and girth at least g. Then $n \geq 3$. We use induction on n; the case $n = 3$ is trivial. Suppose first that G contains a bridge e. Discard e so that G is divided into two connected induced subgraphs G_1 and G_2 on disjoint vertex sets. Let n_i and m_i be the numbers of vertices and edges of G_i, respectively, for $i = 1, 2$. Then $n = n_1 + n_2$ and $m = m_1 + m_2 + 1$. As e is a bridge, at least one of G_1 and G_2 contains a cycle. If both G_1 and G_2 contain cycles, they both have girth at least g, so that by induction

$$m \;=\; m_1 + m_2 + 1 \;\leq\; \frac{g((n_1 - 2) + (n_2 - 2))}{g - 2} + 1 \;<\; \frac{g(n-2)}{g-2}.$$

If, say, G_2 is acyclic, we have $m_2 = n_2 - 1$ and

$$m = m_1 + m_2 + 1 \;\leq\; \frac{g(n_1 - 2)}{g - 2} + n_2 \;<\; \frac{g(n-2)}{g-2}.$$

Finally suppose that G does not contain a bridge. Then each edge of G is contained in exactly two faces. If we denote the number of faces whose border is a cycle consisting of i edges by f_i, we get

$$2m \;=\; \sum_i i f_i \;\geq\; \sum_i g f_i \;=\; gf,$$

as each cycle contains at least g edges. By Theorem 1.5.2, this implies

$$m + 2 = n + f \leq n + \frac{2m}{g} \quad \text{and hence} \quad m \leq \frac{g(n-2)}{g-2}. \qquad \square$$

In particular, we obtain the following immediate consequence of Theorem 1.5.3, since G is either acyclic or has girth at least 3.

Corollary 1.5.4. *Let G be a connected planar graph with n vertices, where $n \geq 3$. Then G contains at most $3n - 6$ edges.* □

Example 1.5.5. By Corollary 1.5.4, the complete graph K_5 is not planar, as a planar graph on five vertices can have at most nine edges. The complete bipartite graph $K_{3,3}$ has girth 4; this graph is not planar by Theorem 1.5.3, as it has more than eight edges.

Exercise 1.5.6. Show that the graphs which arise by omitting one edge e from either K_5 or $K_{3,3}$ are planar. Give plane realizations for $K_5 \setminus e$ and $K_{3,3} \setminus e$ which use straight line segments only.

For the sake of completeness, we will state one of the most famous results in graph theory, namely the characterization of planar graphs due to Kuratowski [Kur30]. We refer the reader to [Har69], [Aig84] or [Tho81] for the elementary but rather lengthy proof. Again we need some definitions. A *subdivision* of a graph G is a graph H which can be derived from G by applying the following operation any number of times: replace an edge $e = ab$ by a path $(a, x_1, \ldots, x_k, b)$, where $x_1, \ldots, x_k$ are an arbitrary number of *new* vertices; that is, vertices which were not in a previous subdivision. For convenience, G is also considered to be a subdivision of itself. Two graphs H and H' are called *homeomorphic* if they are isomorphic to subdivisions of the same graph G. Figure 1.11 shows a subdivision of $K_{3,3}$.

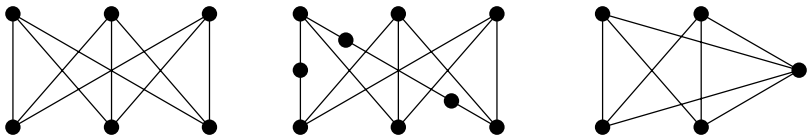


Fig. 1.11. $K_{3,3}$, a subdivision and a contraction

Exercise 1.5.7. Let (V, E) and (V', E') be homeomorphic graphs. Show that $|E| - |V| = |E'| - |V'|$.

Result 1.5.8 (Kuratowski's theorem). *A graph G is planar if and only if it does not contain a subgraph which is homeomorphic to K_5 or $K_{3,3}$.* □

In view of Example 1.5.5, a graph having a subgraph homeomorphic to K_5 or $K_{3,3}$ cannot be planar. For the converse we refer to the sources given above. There is yet another interesting characterization of planarity. If we identify two adjacent vertices u and v in a graph G, we get an *elementary contraction* of G; more precisely, we omit u and v and replace them by a new vertex w which is adjacent to all vertices which were adjacent to u or v before;[8] the

[8] Note that we introduce only one edge wx, even if x was adjacent to *both* u and v, which is the appropriate operation in our context. However, there are occasions

resulting graph is usually denoted by G/e, where $e = uv$. Figure 1.11 also shows a contraction of $K_{3,3}$. A graph G is called *contractible* to a graph H if H arises from G by a sequence of elementary contractions. For the proof of the following theorem see [Wag37], [Aig84], or [HaTu65].

Result 1.5.9 (Wagner's theorem). *A graph G is planar if and only if it does not contain a subgraph which is contractible to K_5 or $K_{3,3}$.*

Exercise 1.5.10. Show that the *Petersen graph* (see Figure 1.12, cf. [Pet98]) is not planar. Give three different proofs using 1.5.3, 1.5.8, and 1.5.9.

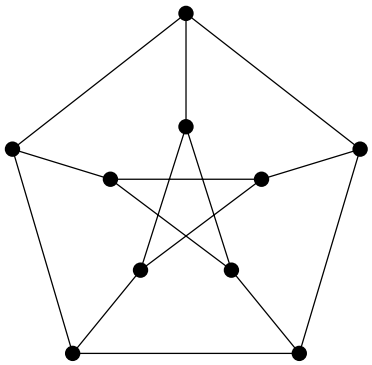


Fig. 1.12. The Petersen graph

Exercise 1.5.11. Show that the Petersen graph is isomorphic to the complement of the triangular graph T_5.

The isomorphisms of a graph G to itself are called *automorphisms*; clearly, they form a group, the *automorphism group* of G. In this book we will not study automorphisms of graphs, except for some comments on Cayley graphs in Chapter 9; we refer the reader to [Yap86], [Har69], or [CaLi91]. However, we give an exercise concerning this topic.

Exercise 1.5.12. Show that the automorphism group of the Petersen graph contains a subgroup isomorphic to the symmetric group S_5. Hint: Use Exercise 1.5.11.

Exercise 1.5.13. What is the minimal number of edges which have to be removed from K_n to get a planar graph? For each n, construct a planar graph having as many edges as possible.

---

where it is actually necessary to introduce two parallel edges wx instead, so that a contracted graph will in general become a multigraph.

The final exercise in this section shows that planar graphs have to contain many vertices of small degree.

Exercise 1.5.14. Let G be a planar graph on n vertices and denote the number of vertices of degree at most d by n_d. Prove

$$n_d \geq \frac{n(d-5)+12}{d+1}$$

and apply this formula to the cases $d = 5$ and $d = 6$. (Hint: Use Corollary 1.5.4.) Can this formula be strengthened?

Much more on planarity (including algorithms) can be found in the monograph by [NiCh88].

1.6 Digraphs

For many applications – especially for problems concerning traffic and transportation – it is useful to give a direction to the edges of a graph, for example to signify a one-way street in a city map. Formally, a *directed graph* or, for short, a *digraph* is a pair $G = (V, E)$ consisting of a finite set V and a set E of ordered pairs (a, b), where $a \neq b$ are elements of V. The elements of V are again called *vertices*, those of E *edges*; the term *arc* is also used instead of *edge* to distinguish between the directed and the undirected case. Instead of $e = (a, b)$, we again write $e = ab$; a is called the *start vertex* or *tail*, and b the *end vertex* or *head* of e. We say that a and b are *incident* with e, and call two edges of the form ab and ba *antiparallel*. To draw a directed graph, we proceed as in the undirected case, but indicate the direction of an edge by an arrow. *Directed multigraphs* can be defined analogously to multigraphs; we leave the precise formulation of the definition to the reader.

There are some operations connecting graphs and digraphs. Let $G = (V, E)$ be a directed multigraph. Replacing each edge of the form (a, b) by an undirected edge $\{a, b\}$, we obtain the *underlying multigraph* $|G|$. Replacing parallel edges in $|G|$ by a single edge, we get the *underlying graph* (G). Conversely, let $G = (V, E)$ be a multigraph. Any directed multigraph H with $|H| = G$ is called an *orientation* of G. Replacing each edge ab in E by two arcs (a, b) and (b, a), we get the *associated directed multigraph* $\overrightarrow{G}$; we also call $\overrightarrow{G}$ the *complete orientation* of G. The complete orientation of K_n is called the *complete digraph* on n vertices. Figure 1.13 illustrates these definitions.

We can now transfer the notions introduced for graphs to digraphs. There are some cases where two possibilities arise; we only look at these cases explicitly and leave the rest to the reader. We first consider trails. Thus let $G = (V, E)$ be a digraph. A sequence of edges $(e_1, \ldots, e_n)$ is called a *trail* if the corresponding sequence of edges in $|G|$ is a trail. We define walks, paths, closed trails and cycles accordingly. Thus, if $(v_0, \ldots, v_n)$ is the corresponding

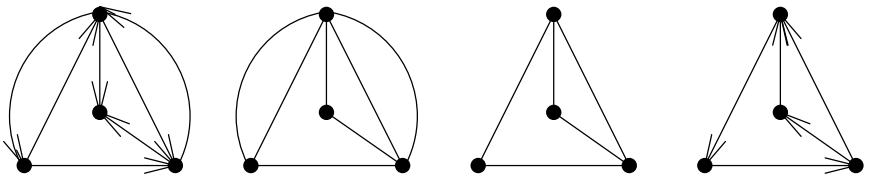


Fig. 1.13. (Directed) multigraphs

sequence of vertices, $v_{i-1}v_i$ or v_iv_{i-1} must be an edge of G. In the first case, we have a *forward edge*, in the second a *backward edge*. If a trail consists of forward edges only, it is called a *directed trail*; analogous definitions can be given for walks, closed trails, etc. In contrast to the undirected case, there may exist directed cycles of length 2, namely cycles of the form (ab, ba).

A *directed Euler tour* in a directed multigraph is a directed closed trail containing each edge exactly once. We want to transfer Euler's theorem to the directed case; this requires some more definitions. The *indegree* $d_{\text{in}}(v)$ of a vertex v is the number of edges with head v, and the *outdegree* $d_{\text{out}}(v)$ of v is the number of edges with tail v. A directed multigraph is called *pseudosymmetric* if $d_{\text{in}}(v) = d_{\text{out}}(v)$ holds for every vertex v. Finally, a directed multigraph G is called *connected* if $|G|$ is connected. We can now state the directed analogue of Euler's theorem. As the proof is quite similar to that of Theorem 1.3.1, we shall leave it to the reader and merely give one hint: the part *(b) implies (c)* needs a somewhat different argument.

Theorem 1.6.1. *Let G be a connected directed multigraph. Then the following statements are equivalent:*

(a) *G has a directed Euler tour.*
(b) *G is pseudosymmetric.*
(c) *The edge set of G can be partitioned into directed cycles.* □

For digraphs there is another obvious notion of connectivity besides simply requiring that the underlying graph be connected. We say that a vertex b of a digraph G is *accessible* from a vertex a if there is a directed walk with start vertex a and end vertex b. As before, we allow walks to have length 0 so that each vertex is accessible from itself. A digraph G is called *strongly connected* if each vertex is accessible from every other vertex. A vertex a from which every other vertex is accessible is called a *root* of G. Thus a digraph is strongly connected if and only if each vertex is a root.

Note that a connected digraph is not necessarily strongly connected. For example, a tree can never be strongly connected; here, of course, a digraph G is called a *tree* if $|G|$ is a tree. If G has a root r, we call G a *directed tree*,

an *arborescence* or a *branching* with root r. Clearly, given any vertex r, an undirected tree has exactly one orientation as a directed tree with root r.

We now consider the question which connected multigraphs can be oriented in such a way that the resulting graph is strongly connected. Such multigraphs are called *orientable*. Thus we ask which connected systems of streets can be made into a system of one-way streets such that people can still move from each point to every other point. The answer is given by the following theorem [Rob39].

Theorem 1.6.2 (Robbins' theorem). *A connected multigraph is orientable if and only if it does not contain any bridge.* □

We will obtain Theorem 1.6.2 by proving a stronger result which allows us to orient the edges one by one, in an arbitrary order. We need some more terminology. A *mixed multigraph* has edges which are either directed or undirected. (We leave the formal definition to the reader.) A *directed trail* in a mixed multigraph is a trail in which each oriented edge is a forward edge, but the trail might also contain undirected edges. A mixed multigraph is called *strongly connected* if each vertex is accessible from every other vertex by a directed trail. The theorem of Robbins is an immediate consequence of the following result due to Boesch and Tindell [BoTi80].

Theorem 1.6.3. *Let G be a mixed multigraph and e an undirected edge of G. Suppose that G is strongly connected. Then e can be oriented in such a way that the resulting mixed multigraph is still strongly connected if and only if e is not a bridge.*

Proof. Obviously, the condition that e is not a bridge is necessary. Thus suppose that e is an undirected edge of G for which neither of the two possible orientations of e gives a strongly connected mixed multigraph. We have to show that e is a bridge of $|G|$. Let u and w be the vertices incident with e, and denote the mixed multigraph we get by omitting e from G by H. Then there is no directed trail in H from u to w: otherwise, we could orient e from w to u and get a strongly connected mixed multigraph. Similarly, there is no directed trail in H from w to u.

Let S be the set of vertices which are accessible from u in H by a directed trail. Then u is, for any vertex $v \in S$, accessible from v in H for the following reason: u is accessible in G from v by a directed trail W; suppose W contains the edge e, then w would be accessible in H from u, which contradicts our observations above. Now put $T = V \setminus S$; as w is in T, this set is not empty. Then every vertex $t \in T$ is accessible from w in H, because t is accessible from w in G, and again: if the trail from w to t in G needed the edge e, then t would be accessible from u in H, and thus t would not be in T.

We now prove that e is the only edge of $|G|$ having a vertex in S and a vertex in T, which shows that e is a bridge. By definition of S, there cannot be an edge (s,t) or an edge $\{s,t\}$ with $s \in S$ and $t \in T$ in G. Finally, if

G contained an edge (t, s), then u would be accessible in H from w, as t is accessible from w and u is accessible from s. □

Mixed multigraphs are an obvious model for systems of streets. However, we will restrict ourselves to multigraphs or directed multigraphs for the rest of this book. One-way streets can be modelled by just using directed multigraphs, and ordinary two-way streets may then be represented by pairs of antiparallel edges.

We conclude this section with a couple of exercises.

Exercise 1.6.4. Let G be a multigraph. Prove that G does not contain a bridge if and only if each edge of G is contained in at least one cycle. (We will see another characterization of these multigraphs in Chapter 7: any two vertices are connected by two edge-disjoint paths.)

Exercise 1.6.5. Let G be a connected graph all of whose vertices have even degree. Show that G has a strongly connected, pseudosymmetric orientation.

Some relevant papers concerning (strongly connected) orientations of graphs are [ChTh78], [ChGT85], and [RoXu88].

Exercise 1.6.6. Prove that any directed closed walk contains a directed cycle. Also show that any directed walk W with start vertex a and end vertex b, where $a \neq b$, contains a directed path from a to b.

Hint: The desired path may be constructed from W by removing directed cycles.

1.7 An application: Tournaments and leagues

We conclude this chapter with an application of the factorizations mentioned before, namely setting up schedules for tournaments[9]. If we want to design a schedule for a tournament, say in soccer or basketball, where each of the $2n$ participating teams should play against each of the other teams exactly once, we can use a factorization $\mathbf{F} = \{F_1, \ldots, F_{2n-1}\}$ of K_{2n}. Then each edge $\{i, j\}$ represents the match between the teams i and j; if $\{i, j\}$ is contained in the factor F_k, this match will be played on the k-th day; thus we have to specify an ordering of the factors. If there are no additional conditions on the schedule, we can use any factorization. At the end of this section we will make a few comments on how to set up *balanced* schedules.

Of course, the above method can also be used to set up a schedule for a league (like, for example, the German soccer league), if we consider the two rounds as two separate tournaments. But then there is the additional problem of planning the home and away games. Look at the first round first. Replace

[9] This section will not be used in the remainder of the book and may be skipped during the first reading.

each 1-factor $F_k \in \mathbf{F}$ by an arbitrary orientation D_k of F_k, so that we get a factorization $\mathbf{D}$ of an orientation of K_{2n} – that is, a tournament as defined in Exercise 7.5.5 below. Then the home and away games of the first round are fixed as follows: if D_k contains the edge ij, the match between the teams i and j will be played on the k-th day of the season as a home match for team i. Of course, when choosing the orientation of the round of return matches, we have to take into account how the first round was oriented; we look at that problem later.

Now one wants home and away games to alternate for each team as far as possible. Hence we cannot just use an arbitrary orientation $\mathbf{D}$ of an arbitrary factorization $\mathbf{F}$ to set up the first round. This problem was solved by de Werra [deW81] who obtained the following results. Define a $(2n \times (2n-1))$-matrix $P = (p_{ik})$ with entries A and H as follows: $p_{ik} = H$ if and only if team i has a home match on the k-th day of the season; that is, if D_k contains an edge of the form ij. De Werra calls this matrix the *home-away pattern* of $\mathbf{D}$. A pair of consecutive entries p_{ik} and $p_{i,k+1}$ is called a *break* if the entries are the same; that is, if there are two consecutive home or away games; thus we want to avoid breaks as far as possible. Before determining the minimal number of breaks, an example might be useful.

Example 1.7.1. Look at the case $n = 3$ and use the factorization of K_6 shown in Figure 1.4; see Exercise 1.1.2. We choose the orientation of the five factors as follows: $D_1 = \{1\infty, 25, 43\}$, $D_2 = \{\infty 2, 31, 54\}$, $D_3 = \{3\infty, 42, 15\}$, $D_4 = \{\infty 4, 53, 21\}$ and $D_5 = \{5\infty, 14, 32\}$. Then we obtain the following matrix P:

$$P = \begin{pmatrix} A & H & A & H & A \\ H & A & H & A & H \\ H & A & A & H & A \\ A & H & H & A & H \\ H & A & H & A & A \\ A & H & A & H & H \end{pmatrix},$$

where the lines and columns are ordered $\infty, 1, \ldots, 5$ and $1, \ldots, 5$, respectively. Note that this matrix contains four breaks, which is best possible for $n = 3$ according to the following lemma.

Lemma 1.7.2. *Every oriented factorization of K_{2n} has at least $2n-2$ breaks.*

Proof. Suppose $\mathbf{D}$ has at most $2n-3$ breaks. Then there are at least three vertices for which the corresponding lines of the matrix P do not contain any breaks. At least two of these lines (the lines i and j, say) have to have the same entry (H, say) in the first column. As both lines do not contain any breaks, they have the same entries, and thus both have the form

$$H \quad A \quad H \quad A \quad H \ldots$$

Then, none of the factors D_k contains one of the edges ij or ji, a contradiction. (In intuitive terms: if the teams i and j both have a home match or both have an away match, they cannot play against each other.) □

The main result of de Werra shows that the bound of Lemma 1.7.2 can always be achieved.

Theorem 1.7.3. *The 1-factorization of K_{2n} given in Exercise 1.1.2 can always be oriented in such a way that the corresponding matrix P contains exactly $2n-2$ breaks.*

Sketch of proof. We give an edge $\{\infty, k\}$ of the 1-factor F_k of Exercise 1.1.2 the orientation $k\infty$ if k is odd, and the orientation ∞k if k is even. Moreover, the edge $\{k+i, k-i\}$ of the 1-factor F_k is oriented as $(k+i, k-i)$ if i is odd, and as $(k-i, k+i)$ if i is even. (Note that the orientation in Example 1.1.3 was obtained using this method.) Then it can be shown that the orientated factorization $\mathbf{D}$ of K_{2n} defined in this way has indeed exactly $2n-2$ breaks. The lines corresponding to the vertices ∞ and 1 do not contain any breaks, whereas exactly one break occurs in all the other lines. The comparatively long, but not really difficult proof of this statement is left to the reader. Alternatively, the reader may consult [deW81] or [deW88]. □

Sometimes there are other properties an optimal schedule should have. For instance, if there are two teams from the same city or region, we might want one of them to have a home game whenever the other has an away game. Using the optimal schedule from Theorem 1.7.3, this can always be achieved.

Corollary 1.7.4. *Let $\mathbf{D}$ be the oriented factorization of K_{2n} with exactly $2n-2$ breaks which was described in Theorem 1.7.3. Then, for each vertex i, there exists a vertex j such that $p_{ik} \neq p_{jk}$ for all $k = 1, \ldots, 2n-1$.*

Proof. The vertex *complementary* to vertex ∞ is vertex 1: team ∞ has a home game on the k-th day of the season (that is, ∞k is contained in D_k) if k is even. Then 1 has the form $1 = k - i$ for some odd i, so that 1 has an away game on that day. Similarly it can be shown that the vertex complementary to $2i$ (for $i = 1, \ldots, n-1$) is the vertex $2i+1$. □

Now we still have the problem of finding a schedule for the return round of the league. Choose oriented factorizations $\mathbf{D}_H$ and $\mathbf{D}_R$ for the first and second round. Of course, we want $\mathbf{D} = \mathbf{D}_H \cup \mathbf{D}_R$ to be a complete orientation of K_{2n}; hence ji should occur as an edge in $\mathbf{D}_R$ if ij occurs in $\mathbf{D}_H$. If this is the case, $\mathbf{D}$ is called a *league schedule* for $2n$ teams. For $\mathbf{D}_H$ and $\mathbf{D}_R$, there are home-away patterns P_H and P_R, respectively; we call $P = (P_H P_R)$ the home-away pattern of $\mathbf{D}$. As before, we want a league schedule to have as few breaks as possible. We have the following result.

Theorem 1.7.5. *Every league schedule $\mathbf{D}$ for $2n$ teams has at least $4n-4$ breaks; this bound can be achieved for all n.*

Proof. As P_H and P_R both have at least $2n-2$ breaks by Lemma 1.7.2, P obviously contains at least $4n-4$ breaks. A league schedule having exactly $4n-4$ breaks can be obtained as follows. By Theorem 1.7.3, there exists an oriented factorization $\mathbf{D}_H = \{D_1, \ldots, D_{2n-1}\}$ of K_{2n} with exactly $2n-2$ breaks. Put $\mathbf{D}_R = \{E_1, \ldots, E_{2n-1}\}$, where E_i is the 1-factor having the opposite orientation as D_{2n-i}; that is, $ji \in E_i$ if and only if $ij \in D_{2n-i}$. Then P_H and P_R each contain exactly $2n-2$ breaks; moreover, the first column of P_R corresponds to the factor E_1, and the last column of P_H corresponds to the factor D_{2n-1} which is the factor with the opposite orientation of E_1. Thus, there are no breaks between these two columns of P, and the total number of breaks is indeed $4n-4$. □

In reality, the league schedules described above are unwelcome, because the return round begins with the same matches with which the first round ended, just with home and away games exchanged. Instead, $\mathbf{D}_R$ is usually defined as follows: $\mathbf{D}_R = \{E_1, \ldots, E_{2n-1}\}$, where E_i is the 1-factor oriented opposite to D_i. Such a league schedule is called *canonical*. The following result can be proved analogously to Theorem 1.7.5.

Theorem 1.7.6. *Every canonical league schedule* $\mathbf{D}$ *for* $2n$ *teams has at least* $6n-6$ *breaks; this bound can be achieved for all* n. □

For more results about league schedules and related problems we refer to [deW80, deW82, deW88] and [Schr80]. In practice, one often has many additional secondary restrictions – sometimes even conditions contradicting each other – so that the above theorems are not sufficient for finding a solution. In these cases, computers are used to look for an adequate solution satisfying the most important requirements. As an example, we refer to [Schr92] who discusses the selection of a schedule for the soccer league in the Netherlands for the season 1988/89. Another actual application with secondary restrictions is treated in [deWJM90], while [GrRo96] contains a survey of some European soccer leagues.

Back to tournaments again! Although any factorization of K_{2n} can be used, in most practical cases there are additional requirements which the schedule should satisfy. Perhaps the teams should play an equal number of times on each of the n playing fields, because these might vary in quality. The best one can ask for in a tournament with $2n-1$ games for each team is, of course, that each team plays twice on each of $n-1$ of the n fields and once on the remaining field. Such a schedule is called a *balanced tournament design*. Every schedule can be written as an $n \times (2n-1)$ matrix $M = (m_{ij})$, where the entry m_{ij} is given by the pair xy of teams playing in round j on field i. Sometimes it is required in addition that, for the first as well as for the last n columns of M, the entries in each row of M form a 1-factor of K_{2n}; this is then called a *partitioned balanced tournament design* (PBTD) on $2n$ vertices. Obviously, such a tournament schedule represents the best possible solution concerning a uniform distribution of the playing fields. We give an example for $n = 5$, and

cite an existence result for PBDT's (without proof) which is due to Lamken and Vanstone [LaVa87, Lam87].

Example 1.7.7. The following matrix describes a PBTD on 10 vertices:

$$\left(\begin{array}{cccc|c|cccc} 94 & 82 & 13 & 57 & 06 & 23 & 45 & 87 & 91 \\ 83 & 95 & 46 & 02 & 17 & 84 & 92 & 05 & 63 \\ 56 & 03 & 97 & 81 & 42 & 67 & 01 & 93 & 85 \\ 12 & 47 & 80 & 96 & 53 & 90 & 86 & 14 & 72 \\ 07 & 16 & 25 & 43 & 98 & 15 & 37 & 26 & 04 \end{array}\right)$$

Result 1.7.8. *Let $n \geq 5$ and $n \notin \{9, 11, 15, 26, 28, 33, 34\}$. Then there exists a PBTD on $2n$ vertices.* □

Finally, we recommend the interesting survey [LaVa89] about tournament designs, which are studied in detail in the books of Anderson [And90, And97].

2

Algorithms and Complexity

If to do were as easy as to know what were good to do...

William Shakespeare

In Theorem 1.3.1 we gave a characterization for Eulerian graphs: a graph G is Eulerian if and only if each vertex of G has even degree. This condition is easy to verify for any given graph. But how can we really find an Euler tour in an Eulerian graph? The proof of Theorem 1.3.1 not only guarantees that such a tour exists, but actually contains a hint how to construct such a tour. We want to convert this hint into a general method for constructing an Euler tour in any given Eulerian graph; in short, into an *algorithm*. In this book we generally look at problems from the algorithmic point of view: we want more than just theorems about existence or structure. As Lüneburg once said [Lue82], it is important in the end that we can compute the objects we are working with. However, we will not go as far as giving concrete programs, but describe our algorithms in a less formal way. Our main goal is to give an overview of the basic methods used in a very large area of mathematics; we can achieve this (without exceeding the limits of this book) only by omitting the details of programming techniques. Readers interested in concrete programs are referred to [SyDK83] and [NiWi78], where programs in PASCAL and FORTRAN, respectively, can be found.

Although many algorithms will occur throughout this book, we will not try to give a formal definition of the concept of algorithms. Such a definition belongs to both mathematical logic and theoretical computer science and is given, for instance, in automata theory or in complexity theory; we refer the reader to [HoUl79] and [GaJo79]. As a general introduction, we also recommend the books [AhHU74, AhHU83].

In this chapter, we will try to show in an intuitive way what an algorithm is and to develop a way to measure the quality of algorithms. In particular, we will consider some basic aspects of graph theoretic algorithms such as, for example, the problem of how to represent a graph. Moreover, we need a way to formulate the algorithms we deal with. We shall illustrate and study these concepts quite thoroughly using two specific examples, namely Euler tours and acyclic digraphs. At the end of the chapter we introduce a class of problems (the so-called NP-complete problems) which plays a central role in

complexity theory; we will meet this type of problem over and over again in this book.

2.1 Algorithms

First we want to develop an intuitive idea what an algorithm is. Algorithms are techniques for solving problems. Here the term *problem* is used in a very general sense: a *problem class* comprises infinitely many *instances* having a common structure. For example, the problem class ET (*Euler tour*) consists of the task to decide – for any given graph G – whether it is Eulerian and, if this is the case, to construct an Euler tour for G. Thus each graph is an instance of ET. In general, an algorithm is a technique which can be used to solve each instance of a given problem class.

According to [BaWo82], an algorithm should have the following properties:

(1) *Finiteness of description*: The technique can be described by a finite text.
(2) *Effectiveness*: Each step of the technique has to be feasible (mechanically) in practice.[1]
(3) *Termination*: The technique has to stop for each instance after a finite number of steps.
(4) *Determinism*: The sequence of steps has to be uniquely determined for each instance.[2]

Of course, an algorithm should also be *correct*, that is, it should indeed solve the problem correctly for each instance. Moreover, an algorithm should be *efficient*, which means it should work as fast and economically as possible. We will discuss this requirement in detail in Sections 2.5 and 2.7.

Note that – like [BaWo82] – we make a difference between an *algorithm* and a *program*: an algorithm is a general technique for solving a problem (that is, it is problem-oriented), whereas a program is the concrete formulation of an algorithm as it is needed for being executed by a computer (and is therefore machine-oriented). Thus, the algorithm may be viewed as the essence of the program. A very detailed study of algorithmic language and program development can be found in [BaWo82]; see also [Wir76].

Now let us look at a specific problem class, namely ET. The following example gives a simple technique for solving this problem for an arbitrary instance, that is, for any given graph.

[1] It is probably because of this aspect of mechanical practicability that some people doubt if algorithms are really a part of mathematics. I think this is a misunderstanding: performing an algorithm in practice does not belong to mathematics, but development and analysis of algorithms – including the translation into a program – do. Like Lüneburg, I am of the opinion that treating a problem algorithmically means understanding it more thoroughly.

[2] In most cases, we will not require this property.

Example 2.1.1. Let G be a graph. Carry out the following steps:

(1) If G is not connected[3] or if G contains a vertex of odd degree, STOP: the problem has no solution.
(2) (We now know that G is connected and that all vertices of G have even degree.) Choose an edge e_1, consider each permutation $(e_2, \ldots, e_m)$ of the remaining edges and check whether $(e_1, \ldots, e_m)$ is an Euler tour, until such a tour is found.

This algorithm is correct by Theorem 1.3.1, but there is still a lot to be said against it. First, it is not really an algorithm in the strict sense, because it does not specify how the permutations of the edges are found and in which order they are examined; of course, this is merely a technical problem which could be dealt with.[4] More importantly, it is clear that examining up to $(m-1)!$ permutations is probably not the most intelligent way of solving the problem. Analyzing the proof of Theorem 1.3.1 (compare also the directed case in 1.6.1) suggests the following alternative technique going back to Hierholzer [Hie73].

Example 2.1.2. Let G be a graph. Carry out the following steps:

(1) If G is not connected or if G contains a vertex of odd degree, STOP: the problem has no solution.
(2) Choose a vertex v_0 and construct a closed trail $C_0 = (e_1, \ldots, e_k)$ as follows: for the end vertex v_i of the edge e_i choose an arbitrary edge e_{i+1} incident with v_i and different from $e_1, \ldots, e_i$, as long as this is possible.
(3) If the closed trail C_i constructed is an Euler tour: STOP.
(4) Choose a vertex w_i on C_i incident with some edge in $E \setminus C_i$. Construct a closed trail Z_i as in (2) (with start and end vertex w_i) in the connected component of w_i in $G \setminus C_i$.
(5) Form a closed trail C_{i+1} by taking the closed trail C_i with start and end vertex w_i and appending the closed trail Z_i. Continue with (3).

This technique yields a correct solution: as each vertex of G has even degree, for any vertex v_i reached in (2), there is an edge not yet used which leaves v_i, except perhaps if $v_i = v_0$. Thus step (2) really constructs a closed trail. In (4), the existence of the vertex w_i follows from the connectedness of G. The above technique is not yet deterministic, but that can be helped by numbering the vertices and edges and – whenever something is to be chosen – always choosing the vertex or edge having the smallest number. In the future, we will not explicitly state how to make such choices deterministically. The steps in 2.1.2 are still rather big; in the first few chapters we will present more detailed versions of the algorithms. Later in the book – when the reader is more used

[3] We can check whether a graph is connected with the BFS technique presented in Section 3.3.

[4] The problem of generating permutations of a given set can be formulated in a graph theoretic way, see Exercise 2.1.3. Algorithms for this are given in [NiWi78] and [Eve73].

to our way of stating algorithms – we will often give rather concise versions of algorithms. A more detailed version of the algorithm in Example 2.1.2 will be presented in Section 2.3.

Exercise 2.1.3. A frequent problem is to order all permutations of a given set in such a way that two subsequent permutations differ by only a transposition. Show that this problem leads to the question whether a certain graph is Hamiltonian. Draw the graph for the case $n = 3$.

Exercise 2.1.4. We want to find out in which cases the closed trail C_0 constructed in Example 2.1.2 (2) is already necessarily Eulerian. An Eulerian graph is called *arbitrarily traceable* from v_0 if each maximal trail beginning in v_0 is an Euler tour; here *maximal* means that all edges incident with the end vertex of the trail occur in the trail. Prove the following results due to Ore (who introduced the concept of arbitrarily traceable graphs [Ore51]) and to [Bae53] and [ChWh70].

(a) G is arbitrarily traceable from v_0 if and only if $G \setminus v_0$ is acyclic.
(b) If G is arbitrarily traceable from v_0, then v_0 is a vertex of maximal degree.
(c) If G is arbitrarily traceable from at least three different vertices, then G is a cycle.
(d) There exist graphs which are arbitrarily traceable from exactly two vertices; one may also prescribe the degree of these vertices.

2.2 Representing graphs

If we want to execute some algorithm for graphs in practice (which usually means on a computer), we have to think first about how to represent a graph. We do this now for digraphs; an undirected graph can then be treated by looking at its complete orientation.[5] Thus let G be a digraph, for example the one shown in Figure 2.1. We have labelled the vertices $1, \ldots, 6$; it is common practice to use $\{1, \ldots, n\}$ as the vertex set of a graph with n vertices. The easiest method to represent G is to list its edges.

Definition 2.2.1 (edge lists).

A directed multigraph G on the vertex set $\{1, \ldots, n\}$ is specified by:

(i) its number of vertices n;
(ii) the list of its edges, given as a sequence of ordered pairs (a_i, b_i), that is, $e_i = (a_i, b_i)$.

The digraph G of Figure 2.1 may then be given as follows.

(i) $n = 6$;

[5] This statement refers only to the representation of graphs in algorithms in general. For each concrete algorithm, we still have to check whether this substitution makes sense. For example, we always get directed cycles by this approach.

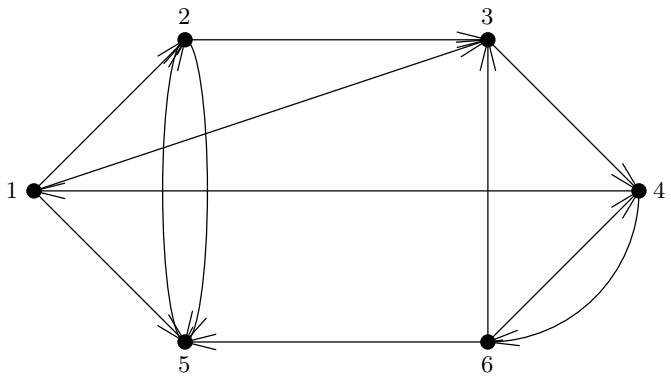


Fig. 2.1. A digraph G

(ii) $12, 23, 34, 15, 52, 65, 46, 64, 41, 63, 25, 13$,

where we write simply ij instead of (i, j). The ordering of the edges was chosen arbitrarily.

A list of m edges can, for example, be implemented by two **arrays** $[1 \dots m]$ (named *head* and *tail*) of type **integer**; in PASCAL we could also define a type **edge** as a **record** of two components of type **integer** and then use an **array**$[1 \dots m]$ of **edge** to store the list of edges.

Lists of edges need little space in memory ($2m$ places for m edges), but they are not convenient to work with. For example, if we need all the vertices adjacent to a given vertex, we have to search through the entire list which takes a lot of time. We can avoid this disadvantage either by ordering the edges in a clever way or by using adjacency lists.

Definition 2.2.2 (incidence lists). A directed multigraph G on the vertex set $\{1, \dots, n\}$ is specified by:

(1) the number of vertices n;

(2) n lists $A_1, \dots, A_n$, where A_i contains the edges beginning in vertex i. Here an edge $e = ij$ is recorded by listing its name and its head j, that is, as the pair (e, j).

The digraph of Figure 2.1 may then be represented as follows:

(1) $n = 6$;

(2) $A_1 : (1, 2), (4, 5), (12, 3)$; $A_2 : (2, 3), (11, 5)$; $A_3 : (3, 4)$; $A_4 : (7, 6), (9, 1)$; $A_5 : (5, 2)$; $A_6 : (6, 5), (8, 4), (10, 3)$,

where we have numbered the edges in the same order as in 2.2.1.

Note that incidence lists are basically the same as edge lists, given in a different ordering and split up into n separate lists. Of course, in the undirected case, each edge occurs now in two of the incidence lists, whereas it would have

been sufficient to put it in the edge list just once. But working with incidence lists is much easier, especially for finding all edges incident with a given vertex. If G is a digraph or a graph (so that there are no parallel edges), it is not necessary to label the edges, and we can use adjacency lists instead of incidence lists.

Definition 2.2.3 (adjacency lists). A digraph with vertex set $\{1, \dots, n\}$ is specified by:

(1) the number of vertices n;
(2) n lists $A_1, \dots, A_n$, where A_i contains all vertices j for which G contains an edge (i, j).

The digraph of Figure 2.1 may be represented by adjacency lists as follows:

(1) $n = 6$;
(2) $A_1 : 2, 3, 5$; $A_2 : 3, 5$; $A_3 : 4$; $A_4 : 1, 6$; $A_5 : 2$; $A_6 : 3, 4, 5$.

In the directed case, we sometimes need all edges with a given end vertex as well as all edges with a given start vertex; then it can be useful to store *backward adjacency lists*, where the end vertices are given, as well. For implementation, it is common to use ordinary or doubly linked lists. Then it is easy to work on all edges in a list consecutively, and to insert or remove edges.

Finally, we give one further method for representing digraphs.

Definition 2.2.4 (adjacency matrices). A digraph G with vertex set $\{1, \dots, n\}$ is specified by an $(n \times n)$-matrix $A = (a_{ij})$, where $a_{ij} = 1$ if and only if (i, j) is an edge of G, and $a_{ij} = 0$ otherwise. A is called the *adjacency matrix* of G. For the digraph of Figure 2.1 we have

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

Adjacency matrices can be implemented simply as an **array** $[1 \dots n, 1 \dots n]$. As they need a lot of space in memory (n^2 places), they should only be used (if at all) to represent digraphs having many edges. Though adjacency matrices are of little practical interest, they are an important theoretical tool for studying digraphs.

Unless stated otherwise, we always represent (directed) multigraphs by incidence or adjacency lists. We will not consider procedures for input or output, or algorithms for treating lists (for operations such as inserting or removing elements, or reordering or searching a list). These techniques are not only used in graph theory but belong to the basic algorithms (searching

and sorting algorithms, fundamental data structures) used in many areas. We refer the reader to the literature, for instance, [AhHU83], [Meh84], and [CoLR90]. We close this section with two exercises about adjacency matrices.

Exercise 2.2.5. Let G be a graph with adjacency matrix A. Show that the (i,k)-entry of the matrix A^h is the number of walks of length h beginning at vertex i and ending at k. Also prove an analogous result for digraphs and directed walks.

Exercise 2.2.6. Let G be a strongly regular graph with adjacency matrix A. Give a quadratic equation for A. Hint: Use Exercise 2.2.5 with $h = 2$.

Examining the adjacency matrix A – and, in particular, the eigenvalues of A – is one of the main tools for studying strongly regular graphs; see [CaLi91]. In general, the eigenvalues of the adjacency matrix of a graph are important in algebraic graph theory; see [Big93] and [ScWi78] for an introduction and [CvDS80, CvDGT87] for a more extensive treatment. Eigenvalues have many noteworthy applications in combinatorial optimization as well; the reader might want to consult the interesting survey [MoPo93].

2.3 The algorithm of Hierholzer

In this section, we study in more detail the algorithm sketched in Example 2.1.2; specifically, we formulate the algorithm of Hierholzer [Hie73] which is able to find an Euler tour in an Eulerian multigraph, respectively a directed Euler tour in a directed Eulerian multigraph. We skip the straightforward checking of the condition on the degrees.

Throughout this book, we will use the symbol $\leftarrow$ for assigning values: $x \leftarrow y$ means that value y is assigned to variable x. Boolean variables can have values *true* and *false*.

Algorithm 2.3.1. Let $G = (V, E)$ be a connected Eulerian multigraph, directed or not, with $V = \{1, \ldots, n\}$. Moreover, let s be a vertex of G. We construct an Euler tour K (which will be directed if G is) with start vertex s.

1. Data structures needed

a) incidence lists $A_1, \ldots, A_n$; for each edge e, we denote the end vertex by end(e);
b) lists K and C for storing sequences of edges forming a closed trail. We use doubly linked lists; that is, each element in the list is linked to its predecessor and its successor, so that these can be found easily;
c) a Boolean mapping *used* on the vertex set, where used(v) has value *true* if v occurs in K and value *false* otherwise, and a list L containing all vertices v for which used(v) = true holds;
d) for each vertex v, a pointer $e(v)$ which is undefined at the start of the algorithm and later points to an edge in K beginning in v;

e) a Boolean mapping *new* on the edge set, where new(e) has value true if e is not yet contained in the closed trail;
f) variables u, v for vertices and e for edges.

2. Procedure TRACE(v, new; C)
The following procedure constructs a closed trail C consisting of edges not yet used, beginning at a given vertex v.

(1) If $A_v = \emptyset$, then **return**.
(2) (Now we are sure that $A_v \neq \emptyset$.) Find the first edge e in A_v and delete e from A_v.
(3) If new(e) = false, go to (1).
(4) (We know that new(e) = true.) Append e to C.
(5) If $e(v)$ is undefined, assign to $e(v)$ the position where e occurs in C.
(6) Assign new(e) $\leftarrow$ false and $v \leftarrow$ end(e).
(7) If used(v) = false, append v to the list L and set used(v) $\leftarrow$ true.
(8) Go to (1).

Here **return** means that the procedure is aborted: one jumps to the end of the procedure, and the execution of the program continues with the procedure which called TRACE. As in the proof of Theorem 1.6.1, the reader may check that the above procedure indeed constructs a closed trail C beginning at v.

3. Procedure EULER($G, s; K$).

(1) $K, L \leftarrow \emptyset$, used($v$) $\leftarrow$ false for all $v \in V$, new(e) $\leftarrow$ true for all $e \in E$.
(2) used(s) $\leftarrow$ true, append s to L.
(3) TRACE(s, new; K);
(4) If L is empty, **return**.
(5) Let u be the last element of L. Delete u from L.
(6) $C \leftarrow \emptyset$.
(7) TRACE(u, new; C).
(8) Insert C in front of $e(u)$ in K.
(9) Go to (4).

In step (3), a maximal closed trail K beginning at s is constructed and all vertices occurring in K are stored in L. In steps (5) to (8) we then try, beginning at the last vertex u of L, to construct a detour C consisting of edges that were not yet used (that is, which have new(e) = true), and to insert this detour into K. Of course, the detour C might be empty. As G is connected, the algorithm ends only if we have used(v) = true for each vertex v of G so that no further detours are possible. If G is a directed multigraph, the algorithm works without the function new; we can then just delete each edge from the incidence list after it has been used.

We close this section with a somewhat lengthy exercise; this requires some definitions. Let S be a given set of s elements, a so-called *alphabet*. Then any finite sequence of elements from S is called a *word* over S. A word of length $N = s^n$ is called a *de Bruijn sequence* if, for each word w of length

n, there exists an index i such that $w = a_i a_{i+1} \dots a_{i+n-1}$, where indices are taken modulo N. For example, 00011101 is a de Bruijn sequence for $s = 2$ and $n = 3$. These sequences take their name from [deB46]. They are closely related to shift register sequences of order n, and are, particularly for $s = 2$, important in coding theory and cryptography; see, for instance, [Gol67], [MacSl77], and [Rue86]; an extensive chapter on shift register sequences can also be found in [Jun93]. We now show how the theorem of Euler for directed multigraphs can be used to construct de Bruijn sequences for all s and n. However, we have to admit loops (a, a) as *edges* here; the reader should convince himself that Theorem 1.6.1 still holds.

Exercise 2.3.2. Define a digraph $G_{s,n}$ having the s^{n-1} words of length $n-1$ over an s-element alphabet S as vertices and the s^n words of length n (over the same alphabet) as edges. The edge $a_1 \dots a_n$ has the word $a_1 \dots a_{n-1}$ as tail and the word $a_2 \dots a_n$ as head. Show that the de Bruijn sequences of length s^n over S correspond to the Euler tours of $G_{s,n}$ and thus prove the existence of de Bruijn sequences for all s and n.

Exercise 2.3.3. Draw the digraph $G_{3,3}$ with $S = \{0, 1, 2\}$ and use Algorithm 2.3.1 to find an Euler tour beginning at the vertex 00; where there is a choice, always choose the smallest edge (smallest when interpreted as a number). Finally, write down the corresponding de Bruijn sequence.

The digraphs $G_{s,n}$ may also be used to determine the number of de Bruijn sequences for given s and n; see Section 4.8. Algorithms for constructing de Bruijn sequences can be found in [Ral81] and [Etz86].

2.4 How to write down algorithms

In this section, we introduce some rules for how algorithms are to be described. Looking again at Algorithm 2.3.1, we see that the structure of the algorithm is not easy to recognize. This is mainly due to the jump commands which hide the loops and conditional ramifications of the algorithm. Here the comments of Jensen and Wirth [JeWi85] about PASCAL should be used as a guideline: "A good rule is to avoid the use of jumps to express regular iterations and conditional execution of statements, for such jumps destroy the reflection of the structure of computation in the textual (static) structures of the program." This motivates us to borrow some notation from PASCAL – even if this language is by now more or less outdated – which is used often in the literature and which will help us to display the structure of an algorithm more clearly. In particular, these conventions emphasize the loops and ramifications of an algorithm. Throughout this book, we shall use the following notation.

Notation 2.4.1 (Ramifications).

if B **then** $P_1; P_2; \ldots; P_k$ **else** $Q_1; Q_2; \ldots; Q_l$ **fi**

is to be interpreted as follows. If condition B is true, the operations $P_1, \ldots, P_k$ are executed; and if B is false, the operations $Q_1, \ldots, Q_l$ are executed. Here the alternative is optional so that we might also have

if B **then** $P_1; P_2; \ldots; P_k$ **fi**

In this case, no operation is executed if condition B is not satisfied.

Notation 2.4.2 (Loops).

for $i = 1$ **to** n **do** $P_1; \ldots, P_k$ **od**

specifies that the operations $P_1, \ldots, P_k$ are executed for each of the (integer) values the *control variable* i takes, namely for $i = 1$, $i = 2$, $\ldots, i = n$. One may also decrement the values of i by writing

for $i = n$ **downto** 1 **do** $P_1; \ldots; P_k$ **od**.

Notation 2.4.3 (Iterations).

while B **do** $P_1; \ldots; P_k$ **od**

has the following meaning. If the condition B holds (that is, if B has Boolean value *true*), the operations $P_1, \ldots, P_k$ are executed, and this is repeated as long as B holds. In contrast,

repeat $P_1; \ldots; P_k$ **until** B

requires first of all to execute the operations $P_1, \ldots, P_k$ and then, if condition B is not yet satisfied, to repeat these operations until finally condition B holds. The main difference between these two ways of describing iterations is that a **repeat** is executed at least once, whereas the operations in a **while** loop are possibly not executed at all, namely if B is not satisfied. Finally,

for $s \in S$ **do** $P_1; \ldots; P_k$ **od**

means that the operations $P_1, \ldots, P_k$ are executed $|S|$ times, once for each element s in S. Here the order of the elements, and hence of the iterations, is not specified.

Moreover, we write **and** for the Boolean operation *and* and **or** for the Boolean operation *or* (not the exclusive or). As before, we shall use $\leftarrow$ for assigning values. The *blocks* of an algorithm arising from ramifications, loops and iterations will be shown by indentations. As an example, we translate the algorithm of Hierholzer into our new notation.

Example 2.4.4. Let G be a connected Eulerian multigraph, directed or not, having vertex set $\{1, \ldots, n\}$. Moreover, let s be a vertex of G. We construct an Euler tour K (which will be directed if G is) with start vertex s. The data structures used are as in 2.3.1. Again, we have two procedures.

Procedure TRACE(v, new; C)

```
(1) while A_v ≠ ∅ do
(2)       delete the first edge e from A_v;
(3)       if new(e) = true
(4)       then append e at the end of C;
(5)            if e(v) is undefined
(6)            then assign the position where e occurs in C to e(v)
(7)            fi
(8)            new(e) ← false, v ← end(e);
(9)            if used(v) = false
(10)           then append v to L;
(11)                used(v) ← true
(12)           fi
(13)      fi
(14) od
```

Procedure EULER($G, s; K$)

```
(1) K ← ∅, L ← ∅;
(2) for v ∈ V do used(v) ← false od
(3) for e ∈ E do new(e) ← true od
(4) used(s) ← true, append s to L;
(5) TRACE(s,new;K);
(6) while L ≠ ∅ do
(7)       let u be the last element of L;
(8)       delete u from L;
(9)       C ← ∅;
(10)      TRACE(u, new; C);
(11)      insert C in front of e(u) in K
(12) od
```

While we need a few more lines than in 2.3.1 to write down the algorithm, the new notation clearly reflects its structure in a much better way. Of course, this is mainly useful if we use a structured language (like PASCAL or C) for programming, but even for programming in a language which depends on jump commands it helps first to understand the structure of the algorithm. We will look at another example in detail in Section 2.6. First, we shall consider the question of how one might judge the quality of algorithms.

2.5 The complexity of algorithms

Complexity theory studies the time and memory space an algorithm needs as a function of on the *size* of the input data; this approach is used to compare different algorithms for solving the same problem. To do this in a formally correct way, we would have to be more precise about what an algorithm is; we

would also have to make clear how input data and the time and space needed by the algorithm are measured. This could be done using Turing machines which were first introduced in [Tur36], but that would lead us too far away from our original intent.

Thus, we will be less formal and simply use the number of vertices or edges of the relevant (directed) multigraph for measuring the size of the input data. The *time complexity* of an algorithm A is the function f, where $f(n)$ is the maximal number of steps A needs to solve a problem instance having input data of length n. The *space complexity* is defined analogously for the memory space needed. We do not specify what a *step* really is, but count the usual arithmetic operations, access to arrays, comparisons, etc. each as one step. This does only make sense if the numbers in the problem do not become really big, which is the case for graph-theoretic problems in practice (but usually not for arithmetic algorithms).

Note that the complexity is always measured for the worst possible case for a given length of the input data. This is not always realistic; for example, most variants of the simplex algorithm in linear programming are known to have exponential complexity although the algorithm works very fast in practice. Thus it might often be better to use some sort of average complexity. But then we would have to set up a probability distribution for the input data, and the whole treatment becomes much more difficult.[6] Therefore, it is common practice to look at the complexity for the worst case.

In most cases it is impossible to calculate the complexity $f(n)$ of an algorithm exactly. We are then content with an estimate of how fast $f(n)$ grows. We shall use the following notation. Let f and g be two mappings from $\mathbb{N}$ to $\mathbb{R}^+$. We write

- $f(n) = O(g(n))$, if there is a constant $c > 0$ such that $f(n) \leq cg(n)$ for all sufficiently large n;
- $f(n) = \Omega(g(n))$, if there is a constant $c > 0$ such that $f(n) \geq cg(n)$ for all sufficiently large n;
- $f(n) = \Theta(g(n))$, if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

If $f(n) = \Theta(g(n))$, we say that f has *rate of growth* $g(n)$. If $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$, then f has at most or at least rate of growth $g(n)$, respectively. If the time or space complexity of an algorithm is $O(g(n))$, we say that the algorithm has complexity $O(g(n))$.

We will usually consider the time complexity only and just talk of the *complexity*. Note that the space complexity is at most as large as the time complexity, because the data taking up memory space in the algorithm have to be read first.

Example 2.5.1. For a graph G we obviously have $|E| = O(|V|^2)$; if G is connected, Theorem 1.2.6 implies that $|E| = \Omega(|V|)$. Graphs with $|E| = \Theta(|V|^2)$

[6] How difficult it really is to deal with such a distribution can be seen in the probabilistic analysis of the simplex algorithm, cf. [Bor87].

are often called *dense*, while graphs with $|E| = \Theta(|V|)$ are called *sparse*. Corollary 1.5.4 tells us that the connected planar graphs are sparse. Note that $O(\log|E|)$ and $O(\log|V|)$ are the same for connected graphs, because the logarithms differ only by a constant factor.

Example 2.5.2. Algorithm 2.3.1 has complexity $\Theta(|E|)$, because each edge is treated at least once and at most twice during the procedure TRACE; each such examination of an edge is done in a number of steps bounded by a constant, and constants can be disregarded in the notation we use. Note that $|V|$ does not appear because of $|E| = \Omega(|V|)$, as G is connected.

If, for a problem P, there exists an algorithm having complexity $O(f(n))$, we say that P has complexity at most $O(f(n))$. If each algorithm for P has complexity $\Omega(g(n))$, we say that P has complexity at least $\Omega(g(n))$. If, in addition, there is an algorithm for P with complexity $O(g(n))$, then P has complexity $\Theta(g(n))$.

Example 2.5.3. The problem of finding Euler tours has complexity $\Theta(|E|)$: we have provided an algorithm with this complexity, and obviously each algorithm for this problem has to consider all the edges to be able to put them into a sequence forming an Euler tour.

Unfortunately, in most cases it is much more difficult to find lower bounds for the complexity of a problem than to find upper bounds, because it is hard to say something non-trivial about all possible algorithms for a problem. Another problem with the above conventions for the complexity of algorithms lies in disregarding constants, as this means that the rates of growth are only asymptotically significant – that is, for very large n. For example, if we know that the rate of growth is linear – that is $O(n)$ – but the constant is $c = 1,000,000$, this would not tell us anything about the common practical cases involving relatively small n. In fact, the asymptotically fastest algorithms for integer multiplication are only interesting in practice if the numbers treated are quite large; see, for instance, [AhHU74]. However, for the algorithms we are going to look at, the constants will always be small (mostly ≤ 10).

In practice, the *polynomial* algorithms – that is, the algorithms of complexity $O(n^k)$ for some k – have proved to be the most useful. Such algorithms are also called *efficient* or – following Edmonds [Edm65b] – *good*. Problems for which a polynomial algorithm exists are also called *easy*, whereas problems for which no polynomial algorithm can exist are called *intractable* or *hard*. This terminology may be motivated by considering the difference between polynomial and exponential rates of growth. This difference is illustrated in Table 2.1 and becomes even more obvious by thinking about the consequences of improved technology. Suppose we can at present – in some fixed amount of time, say an hour – solve an instance of size N on a computer, at rate of growth $f(n)$. What effect does a 1000-fold increase in computer speed then have on the size of instances we are able to solve? If $f(n)$ is polynomial, say

n^k, we will be able to solve an instance of size cN, where $c = 10^{3/k}$; for example, if $k = 3$, this still means a factor of $c = 10$. If the rate of growth is exponential, say a^c, there is only an improvement of constant size: we will be able to solve instances of size $N + c$, where $a^c = 1000$. For example, if $a = 2$, we have $c \approx 9.97$; for $a = 5$, $c \approx 4.29$.

Table 2.1. Rates of growth

$f(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 50$	$n = 100$
n	10	20	30	50	100
n^2	100	400	900	2,500	10,000
n^3	1,000	8,000	27,000	125,000	1,000,000
n^4	10,000	160,000	810,000	6,250,000	100,000,000
2^n	1,024	1,048,576	$\approx 10^9$	$\approx 10^{15}$	$\approx 10^{30}$
5^n	9,765,625	$\approx 10^{14}$	$\approx 10^{21}$	$\approx 10^{35}$	$\approx 10^{70}$

We see that, from a practical point of view, it makes sense to consider a problem well solved only when we have found a polynomial algorithm for it. Moreover, if there is a polynomial algorithm, in many cases there is even an algorithm of rate of growth n^k with $k \leq 3$. Unfortunately, there is a very large class of problems, the so-called *NP-complete* problems, for which not only is no polynomial algorithm known, but there is good reason to believe that such an algorithm cannot exist. These questions are investigated more thoroughly in complexity theory; see [GaJo79] or [Pap94]. Most algorithms we study in this book are polynomial. Nevertheless, we will explain in Section 2.7 what NP-completeness is, and show in Section 2.8 that determining a Hamiltonian cycle and the TSP are such problems. In Chapter 15, we will develop strategies for solving such problems (for example, approximation or complete enumeration) using the TSP as an example; actually, the TSP is often used as the standard example for NP-complete problems. We will encounter quite a few NP-complete problems in various parts of this book.

It has to be admitted that most problems arising from practice tend to be NP-complete. It is indeed rare to be able to solve a practical problem just by applying one of the polynomial algorithms we shall treat in this book. Nevertheless, these algorithms are also very important, since they are regularly used as sub-routines for solving more involved problems.

2.6 Directed acyclic graphs

In this section, we provide another illustration for the definitions and notation introduced in the previous sections by considering an algorithm which deals with directed *acyclic* graphs, that is, digraphs which do not contain directed closed trails. This sort of graph occurs in many applications, for example in the

planning of projects (see 3.7) or for representing the structure of arithmetic expressions having common parts, see [AhHU83]. First we give a mathematical application.

Example 2.6.1. Let $(M, \preceq)$ be a *partially ordered set*, for short, a *poset*. This is a set M together with a reflexive, antisymmetric and transitive relation $\preceq$. Note that M corresponds to a directed graph G having vertex set M and the pairs (x, y) with $x \prec y$ as edges; because of transitivity, G is acyclic.

A common problem is to check whether a given directed graph is acyclic and, if this is the case, to find a *topological sorting* of its vertices. That is, we require an enumeration of the vertices of G (labelling them with the numbers $1, \ldots, n$, say) such that $i < j$ holds for each edge ij. Using the following lemma, we shall show that such a sorting exists for every directed acyclic graph.

Lemma 2.6.2. *Let G be a directed acyclic graph. Then G contains at least one vertex with $d_{\text{in}}(v) = 0$.*

Proof. Choose a vertex v_0. If $d_{\text{in}}(v_0) = 0$, there is nothing to show. Otherwise, there is an edge v_1v_0. If $d_{\text{in}}(v_1) = 0$, we are done. Otherwise, there exists an edge v_2v_1. As G is acyclic, $v_2 \neq v_0$. Continuing this procedure, we get a sequence of distinct vertices $v_0, v_1, \ldots, v_k, \ldots$. As G has only finitely many vertices, this sequence has to terminate, so that we reach a vertex v with $d_{\text{in}}(v) = 0$. □

Theorem 2.6.3. *Every directed acyclic graph admits a topological sorting.*

Proof. By Lemma 2.6.2, we may choose a vertex v with $d_{\text{in}}(v) = 0$. Consider the directed graph $H = G \setminus v$. Obviously, H is acyclic as well and thus can be sorted topologically, using induction on the number of vertices, say by labelling the vertices as $v_2, \ldots, v_n$. Then $(v, v_2, \ldots, v_n)$ is the desired topological sorting of G. □

Corollary 2.6.4. *Each partially ordered set may be embedded into a linearly ordered set.*

Proof. Let $(v_1, \ldots, v_n)$ be a topological sorting of the corresponding directed acyclic graph. Then $v_i \prec v_j$ always implies $i < j$, so that $v_1 \prec \ldots \prec v_n$ is a complete linear ordering. □

Next we present an algorithm which decides whether a given digraph is acyclic and, if this is the case, finds a topological sorting. We use the same technique as in the proof of Theorem 2.6.3, that is, we successively delete vertices with $d_{\text{in}}(v) = 0$. To make the algorithm more efficient, we use a list of the indegrees $d_{\text{in}}(v)$ and bring it up to date whenever a vertex is deleted; in this way, we do not have to search the entire graph to find vertices with indegree 0. Moreover, we keep a list of all the vertices having $d_{\text{in}}(v) = 0$. The following algorithm is due to Kahn [Kah62].

Algorithm 2.6.5. Let G be a directed graph with vertex set $\{1, \ldots, n\}$. The algorithm checks whether G is acyclic; in this case, it also determines a topological sorting.

Data structures needed

a) adjacency lists $A_1, \ldots, A_n$;
b) a function *ind*, where $\mathrm{ind}(v) = d_{in}(v)$;
c) a function *topnr*, where $\mathrm{topnr}(v)$ gives the index of vertex v in the topological sorting;
d) a list L of the vertices v having $\mathrm{ind}(v) = 0$;
e) a Boolean variable *acyclic* and an integer variable N (for counting).

Procedure TOPSORT (G; topnr,acyclic)

(1) $N \leftarrow 1$, $L \leftarrow \emptyset$;
(2) **for** $i = 1$ **to** n **do** $\mathrm{ind}(i) \leftarrow 0$ **od**
(3) **for** $i = 1$ **to** n **do**
(4) **for** $j \in A_i$ **do** $\mathrm{ind}(j) \leftarrow \mathrm{ind}(j) + 1$ **od**
(5) **od**
(6) **for** $i = 1$ **to** n **do if** $\mathrm{ind}(i) = 0$ **then** append i to L **fi od**
(7) **while** $L \neq \emptyset$ **do**
(8) delete the first vertex v from L;
(9) $\mathrm{topnr}(v) \leftarrow N$; $N \leftarrow N + 1$;
(10) **for** $w \in A_v$ **do**
(11) $\mathrm{ind}(w) \leftarrow \mathrm{ind}(w) - 1$;
(12) **if** $\mathrm{ind}(w) = 0$ **then** append w to L **fi**
(13) **od**
(14) **od**
(15) **if** $N = n + 1$ **then** acyclic $\leftarrow$ true **else** acyclic $\leftarrow$ false **fi**

Theorem 2.6.6. *Algorithm 2.6.5 determines whether G is acyclic and constructs a topological sorting if this is the case; the complexity is $O(|E|)$ provided that G is connected.*

Proof. The discussion above shows that the algorithm is correct. As G is connected, we have $|E| = \Omega(|V|)$, so that initializing the function ind and the list L in step (2) and (6), respectively, does not take more than $O(|E|)$ steps. Each edge is treated exactly once in step (4) and at most once in step (10) which shows that the complexity is $O(|E|)$. □

When checking whether a directed graph is acyclic, each edge has to be treated at least once. This observation immediately implies the following result.

Corollary 2.6.7. *The problem of checking whether a given connected digraph is acyclic or not has complexity $\Theta(|E|)$.* □

Exercise 2.6.8. Show that any algorithm which checks whether a digraph given in terms of its adjacency matrix is acyclic or not has complexity at least $\Omega(|V|^2)$.

The above exercise shows that the complexity of an algorithm might depend considerably upon the chosen representation for the directed multigraph.

Exercise 2.6.9. Apply Algorithm 2.6.5 to the digraph G in Figure 2.2, and give an alternative drawing for G which reflects the topological ordering.

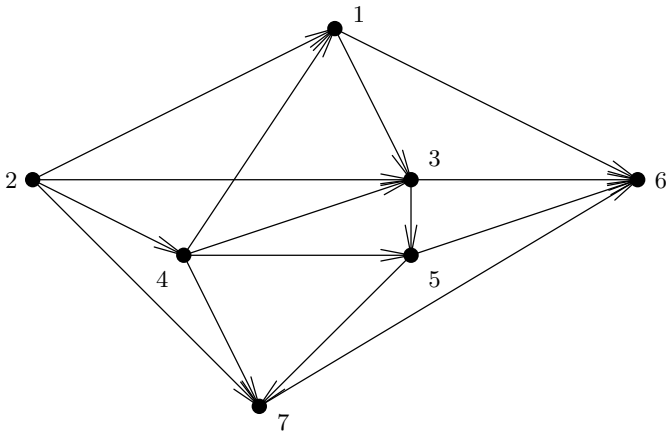


Fig. 2.2. A digraph

In the remainder of this book, we will present algorithms in less detail. In particular, we will not explain the data structures used explicitly if they are clear from the context. Unless stated otherwise, all multigraphs will be represented by incidence or adjacency lists.

2.7 NP-complete problems

Up to now, we have encountered only polynomial algorithms; problems which can be solved by such an algorithm are called *polynomial* or – as in Section 2.5 – *easy*. Now we turn our attention to another class of problems. To do so, we restrict ourselves to *decision problems*, that is, to problems whose solution is either *yes* or *no*. The following problem HC is such a problem; other decision problems which we have solved already are the question whether a given multigraph (directed or not) is Eulerian, and the problem whether a given digraph is acyclic.

Problem 2.7.1 (Hamiltonian cycle, HC). Let G be a given connected graph. Does G have a Hamiltonian cycle?

We will see that Problem 2.7.1 is just as difficult as the TSP defined in Problem 1.4.9. To do so, we have to make an excursion into complexity theory. The following problem is arguably the most important decision problem.

Problem 2.7.2 (satisfiability, SAT). Let $x_1, \ldots, x_n$ be *Boolean variables*: they take values *true* or *false*. We consider formulae in $x_1, \ldots, x_n$ in *conjunctive normal form*, namely terms $C_1 C_2 \ldots C_m$, where each of the C_i has the form $x'_i + x'_j + \ldots$ with $x'_i = x_i$ or $x'_i = \overline{x_i}$; in other words, each C_i is a disjunction of some, possibly negated, variables.[7] The problem requires deciding whether any of the possible combinations of values for the x_i gives the entire term $C_1 \ldots C_m$ the value *true*. In the special case where each of the C_i consists of exactly three literals, the problem is called 3-*satisfiability* (3-SAT).

Most of the problems of interest to us are not decision problems but *optimization problems*: among all possible structures of a given kind (for example, for the TSP considered in Section 1.4, among all possible tours), we look for the optimal one with respect to a certain criterion (for example, for the shortest tour). We shall solve many such problems: finding shortest paths, minimal spanning trees, maximal flows, maximal matchings, etc.

Note that each optimization problem gives rise to a decision problem involving an additional parameter; we illustrate this using the TSP. For a given matrix $W = (w_{ij})$ and every positive integer M, the associated decision problem is the question whether there exists a tour π such that $w(\pi) \leq M$. There is a further class of problems lying in between decision problems and optimization problems, namely *evaluation problems*; here one asks for the value of an optimal solution without requiring the explicit solution itself. For example, for the TSP we may ask for the length of an optimal tour without demanding to be shown this tour. Clearly, every algorithm for an optimization problem solves the corresponding evaluation problem as well; similarly, solving an evaluation problems also gives a solution for the associated decision problem. It is not so clear whether the converse of these statements is true. But surely an optimization problem is at least as hard as the corresponding decision problem, which is all we will need to know.[8]

[7] We write $\overline{p}$ for the negation of the logical variable p, $p + q$ for the disjunction p *or* q, and pq for the conjunction p *and* q. The x'_i are called *literals*, the C_i are *clauses*.

[8] We may solve an evaluation problem quite efficiently by repeated calls of the associated decision problem, if we use a binary search. But in general, we do not know how to find an optimal solution just from its value. However, in problems from graph theory, it is often sufficient to know that the value of an optimal solution can be determined polynomially. For example, for the TSP we would check in polynomial time whether there is an optimal solution not containing a given edge. In this way we can find an optimal tour by sequentially using the algorithm for the evaluation problem a linear number of times.

We denote the class of all polynomial decision problems by P (for *polynomial*).[9] The class of decision problems for which a positive answer can be verified in polynomial time is denoted by NP (for *non-deterministic polynomial*). That is, for an NP-problem, in addition to the answer *yes* or *no* we require the specification of a *certificate* enabling us to verify the correctness of a positive answer in polynomial time. We explain this concept by considering two examples, first using the TSP. If a possible solution – for the TSP, a tour – is presented, it has to be possible to check in polynomial time

- whether the candidate has the required structure (namely, whether it is really a tour, and not, say, just a permutation with several cycles)
- and whether the candidate satisfies the condition imposed (that is, whether the tour has length $w(\pi) \leq M$, where M is the given bound).

Our second example is the question whether a given connected graph is not Eulerian. A positive answer can be verified by giving a vertex of odd degree.[10] We emphasize that the definition of NP does not demand that a negative answer can be verified in polynomial time. The class of decision problems for which a negative answer can be verified in polynomial time is denoted by Co-NP.[11]

Obviously, P $\subset$ NP $\cap$ Co-NP, as any polynomial algorithm for a decision problem even provides the correct answer in polynomial time. On the other hand, it is not clear whether every problem from NP is necessarily in P or in Co-NP. For example, we do not know any polynomial algorithm for the TSP. Nevertheless, we can verify a positive answer in polynomial time by checking whether the certificate π is a cyclic permutation of the vertices, calculating $w(\pi)$, and comparing $w(\pi)$ with M. However, we do not know any polynomial algorithm which could check a negative answer for the TSP, namely the assertion that no tour of length $\leq M$ exists (for an arbitrary M). In fact, the questions whether P = NP or NP = Co-NP are the outstanding questions of complexity theory. As we will see, there are good reasons to believe that the conjecture P $\neq$ NP (and NP $\neq$ Co-NP) is true. To this end, we consider a special class of problems within NP.

A problem is called *NP-complete* if it is in NP and if the polynomial solvability of this problem would imply that all other problems in NP are solvable in polynomial time as well. More precisely, we require that any given

[9] To be formally correct, we would have to state how an instance of a problem is *coded* (so that the length of the input data could be measured) and what an *algorithm* is. This can be done by using the concept of a Turing machine introduced by [Tur36]. For detailed expositions of complexity theory, we refer to [GaJo79], [LePa81], and [Pap94].

[10] Note that no analogous certificate is known for the question whether a graph is not Hamiltonian.

[11] Thus, for NP as well as for Co-NP, we look at a kind of *oracle* which presents some (positive or negative) answer to us; and this answer has to be verifiable in polynomial time.

problem in NP can be transformed in polynomial time to the specific problem such that a solution of this NP-complete problem also gives a solution of the other, arbitrary problem in NP. We will soon see some examples of such transformations. Note that NP-completeness is a very strong condition: if we could find a polynomial algorithm for such a problem, we would prove P = NP. Of course, there is no obvious reason why any NP-complete problems should exist. The following celebrated theorem due to Cook [Coo71] provides a positive answer to this question; for the rather technical and lengthy proof, we refer to [GaJo79] or [PaSt82].

Result 2.7.3 (Cook's theorem). *SAT and 3-SAT are NP-complete.* □

Once a first NP-complete problem (such as 3-SAT) has been found, other problems can be shown to be NP-complete by transforming the known NP-complete problem in polynomial time to these problems. Thus it has to be shown that a polynomial algorithm for the new problem implies that the given NP-complete problem is polynomially solvable as well. As a major example, we shall present a (quite involved) polynomial transformation of 3-SAT to HC in Section 2.8. This will prove the following result of Karp [Kar72] which we shall use right now to provide a rather simple example for the method of transforming problems.

Theorem 2.7.4. *HC is NP-complete.* □

Theorem 2.7.5. *TSP is NP-complete.*

Proof. We have already seen that TSP is in NP. Now assume the existence of a polynomial algorithm for TSP. We use this hypothetical algorithm to construct a polynomial algorithm for HC as follows. Let $G = (V, E)$ be a given connected graph, where $V = \{1, \ldots, n\}$, and let K_n be the complete graph on V with weights

$$w_{ij} := \begin{cases} 1 & \text{for } ij \in E, \\ 2 & \text{otherwise.} \end{cases}$$

Obviously, G has a Hamiltonian cycle if and only if there exists a tour π of weight $w(\pi) \leq n$ (and then, of course, $w(\pi) = n$) in K_n. Thus the given polynomial algorithm for TSP allows us to decide HC in polynomial time; hence Theorem 2.7.4 shows that TSP is NP-complete. □

Exercise 2.7.6 (directed Hamiltonian cycle, DHC). Show that it is NP-complete to decide whether a directed graph G contains a directed Hamiltonian cycle.

Exercise 2.7.7 (Hamiltonian path, HP). Show that it is NP-complete to decide whether a given graph G contains a *Hamiltonian path* (that is, a path containing each vertex of G).

Exercise 2.7.8 (Longest path). Show that it is NP-complete to decide whether a given graph G contains a path consisting of at least k edges. Prove that this also holds when we are allowed to specify the end vertices of the path. Also find an analogous results concerning longest cycles.

Hundreds of problems have been recognized as NP-complete, including many which have been studied for decades and which are important in practice. Detailed lists can be found in [GaJo79] or [Pap94]. For none of these problems a polynomial algorithm could be found in spite of enormous efforts, which gives some support for the conjecture P $\neq$ NP.[12] In spite of some theoretical progress, this important problem remains open, but at least it has led to the development of *structural complexity theory*; see, for instance, [Boo94] for a survey. Anyway, proving that NP-complete problems are indeed hard would not remove the necessity of dealing with these problems in practice. Some possibilities how this might be done will be discussed in Chapter 15.

Finally, we introduce one further notion. A problem which is not necessarily in NP, but whose polynomial solvability would nevertheless imply P = NP is called *NP-hard*. In particular, any optimization problem corresponding to an NP-complete decision problem is an NP-hard problem.

2.8 HC is NP-complete

In this section (which is somewhat technical and may be skipped during the first reading) we prove Theorem 2.7.4 and show that HC is NP-complete. Following [GaJo79], our proof makes a detour via another very important NP-complete graph theoretical problem; a proof which transforms 3-SAT directly to HC can be found in [PaSt82]. First, a definition. A *vertex cover* of a graph $G = (V, E)$ is a subset V' of V such that each edge of G is incident with at least one vertex in V'.

Problem 2.8.1 (vertex cover, VC). Let $G = (V, E)$ be a graph and k a positive integer. Does G have a vertex cover V' with $|V'| \leq k$?

Obviously, the problem VC is in NP. We prove a further important result due to Karp [Kar72] and show that VC is NP-complete by transforming 3-SAT polynomially to VC and applying Result 2.7.3. The technique we employ is used often for this kind of proof: we construct, for each instance of 3-SAT, a graph consisting of *special-purpose components* combined in an elaborate way. This strategy should become clear during the proofs of Theorem 2.8.2 and Theorem 2.7.4.

Theorem 2.8.2. *VC is NP-complete.*

[12] Thus we can presumably read *NP-complete* also as *non-polynomial*. However, one also finds the opposite conjecture P = NP (along with some incorrect attempts at proving this claim) and the suggestion that the problem might be undecidable.

Proof. We want to transform 3-SAT polynomially to VC. Thus let $C_1 \dots C_m$ be an instance of 3-SAT, and let $x_1, \dots, x_n$ be the variables occurring in $C_1, \dots, C_m$. For each x_i, we form a copy of the complete graph K_2:

$$T_i = (V_i, E_i) \quad \text{where} \quad V_i = \{x_i, \overline{x_i}\} \quad \text{and} \quad E_i = \{x_i\overline{x_i}\}.$$

The purpose of these *truth-setting components* is to determine the Boolean value of x_i. Similarly, for each clause C_j $(j = 1, \dots, m)$, we form a copy $S_j = (V'_j, E'_j)$ of K_3:

$$V'_j = \{c_{1j}, c_{2j}, c_{3j}\} \quad \text{and} \quad E'_j = \{c_{1j}c_{2j}, c_{1j}c_{3j}, c_{2j}c_{3j}\}.$$

The purpose of these *satisfaction-testing components* is to check the Boolean value of the clauses. The $m+n$ graphs constructed in this way are the special-purpose components of the graph G which we will associate with $C_1 \dots C_m$; note that they merely depend on n and m, but not on the specific structure of $C_1 \dots C_m$. We now come to the only part of the construction of G which uses the specific structure, namely connecting the S_j and the T_i by further edges, the *communication edges*. For each clause C_j, we let u_j, v_j, and w_j be the three literals occurring in C_j and define the following set of edges:

$$E''_j = \{c_{1j}u_j, c_{2j}v_j, c_{3j}w_j\}.$$

Finally, we define $G = (V, E)$ as the union of all these vertices and edges:

$$V := \bigcup_{i=1}^{n} V_i \cup \bigcup_{j=1}^{m} V'_j \qquad \text{and} \qquad E := \bigcup_{i=1}^{n} E_i \cup \bigcup_{j=1}^{m} E'_j \cup \bigcup_{j=1}^{m} E''_j.$$

Clearly, the construction of G can be performed in polynomial time in n and m. Figure 2.3 shows, as an example, the graph corresponding to the instance

$$(x_1 + \overline{x_3} + \overline{x_4})(\overline{x_1} + x_2 + \overline{x_4})$$

of 3-SAT. We now claim that G has a vertex cover W with $|W| \leq k = n + 2m$ if and only if there is a combination of Boolean values for $x_1, \dots, x_n$ such that $C_1 \dots C_m$ has value *true*.

First, let W be such a vertex cover. Obviously, each vertex cover of G has to contain at least one of the two vertices in V_i (for each i) and at least two of the three vertices in V'_j (for each j), since we have formed complete subgraphs on these vertex sets. Thus W contain at least $n + 2m = k$ vertices, and hence actually $|W| = k$. But then W has to contain exactly one of the two vertices x_i and $\overline{x_i}$ and exactly two of the three vertices in S_j, for each i and for each j. This fact allows us to use W to define a combination w of Boolean values for the variables $x_1, \dots, x_n$ as follows. If W contains x_i, we set $w(x_i) = \text{true}$; otherwise W has to contain the vertex $\overline{x_i}$, and we set $w(x_i) = \text{false}$.

Now consider an arbitrary clause C_j. As W contains exactly two of the three vertices in V'_j, these two vertices are incident with exactly two of the

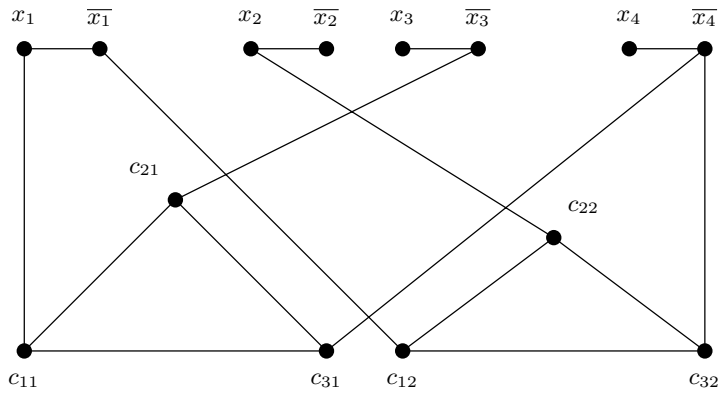


Fig. 2.3. An instance of VC

three edges in E_j''. As W is a vertex cover, it has to contain a vertex incident with the third edge, say $c_{3j}w_j$, and hence W contains the corresponding vertex in one of the V_i – here the vertex corresponding to the literal w_j, that is, to either x_i or $\overline{x_i}$. By our definition of the truth assignment w, this literal has the value *true*, making the clause C_j true. As this holds for all j, the formula $C_1 \dots C_m$ also takes the Boolean value *true* under w.

Conversely, let w be an assignment of Boolean values for the variables $x_1, \dots, x_n$ such that $C_1 \dots C_m$ takes the value *true*. We define a subset $W \subset V$ as follows. If $w(x_i) =$ true, W contains the vertex x_i, otherwise W contains $\overline{x_i}$ (for $i = 1, \dots, n$). Then all edges in E_i are covered. Moreover, at least one edge e_j of E_j'' is covered (for each $j = 1, \dots, m$), since the clause C_j takes the value *true* under w. Adding the end vertices in S_j of the other two edges of E_j'' to W, we cover all edges of E_j'' and of E_j' so that W is indeed a vertex cover of cardinality k. □

Exercise 2.8.3. An *independent set (IS)* (or *stable set*) in a graph $G = (V, E)$ is a subset U of the vertex set V such that no two vertices in U are adjacent. A *clique* in G is a subset C of V such that all pairs of vertices in C are adjacent. Prove that the following two problems are NP-complete by relating them to the problem VC.

(a) **Independent set.** Does a given graph G contain an independent set of cardinality $\geq k$?

(b) **Clique.** Does a given graph G contain a clique of cardinality $\geq k$?

In view of from Theorem 2.8.2, we may now prove the NP-completeness of HC by transforming VC polynomially to HC; as before, we follow [GaJo79]. Let $G = (V, E)$ be a given instance of VC, and k a positive integer. We have to construct a graph $G' = (V', E')$ in polynomial time such that G' is Hamiltonian if and only if G has a vertex cover of cardinality at most k. Again,

we first define some special-purpose components. There are k special vertices $a_1, \ldots, a_k$ called *selector vertices*, as they will be used to select k vertices from V. For each edge $e = uv \in E$, we define a subgraph $T_e = (V'_e, E'_e)$ with 12 vertices and 14 edges as follows (see Figure 2.4):

$$V'_e := \{(u,e,i) : i = 1,\ldots,6\} \cup \{(v,e,i) : i = 1,\ldots,6\};$$

$$\begin{aligned} E'_e := {} & \{\{(u,e,i),(u,e,i+1)\} : i = 1,\ldots,5\} \\ & \cup \{\{(v,e,i),(v,e,i+1)\} : i = 1,\ldots,5\} \\ & \cup \{\{(u,e,1),(v,e,3)\},\{(u,e,3),(v,e,1)\}\} \\ & \cup \{\{(u,e,4),(v,e,6)\},\{(u,e,6),(v,e,4)\}\}. \end{aligned}$$

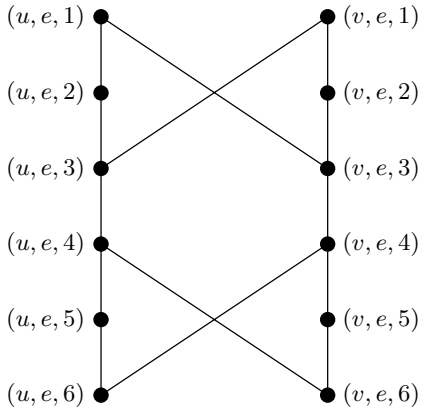


Fig. 2.4. Cover-testing component

This *cover-testing component* T_e will make sure that the vertex set $W \subset V$ determined by the selectors $a_1, \ldots, a_k$ contains at least one of the vertices incident with e. Only the *outer* vertices $(u,e,1), (u,e,6), (v,e,1)$ and $(v,e,6)$ of T_e will be incident with further edges of G'; this forces each Hamiltonian cycle of G' to run through each of the subgraphs T_e using one of the paths shown in Figure 2.5, as the reader can (and should) easily check.

Now we describe the remaining edges of G'. For each vertex $v \in V$, we label the edges incident with v as $ev_1, \ldots, ev_{\deg v}$ and connect the $\deg v$ corresponding graphs T_{ev_i} by the following edges:

$$E'_v := \{\{(v, ev_i, 6), (v, ev_{i+1}, 1)\} : i = 1, \ldots, \deg v - 1\}.$$

These edges create a path in G' which contains precisely the vertices (x,y,z) with $x = v$, see Figure 2.6. Finally, we connect the start and end vertices of all these paths to each of the selectors a_j:

$$E'' := \{\{a_j, (v, ev_1, 1)\} : j = 1, \ldots, k\} \cup \{\{a_j, (v, ev_{\deg v}, 6)\} : j = 1, \ldots k\}.$$

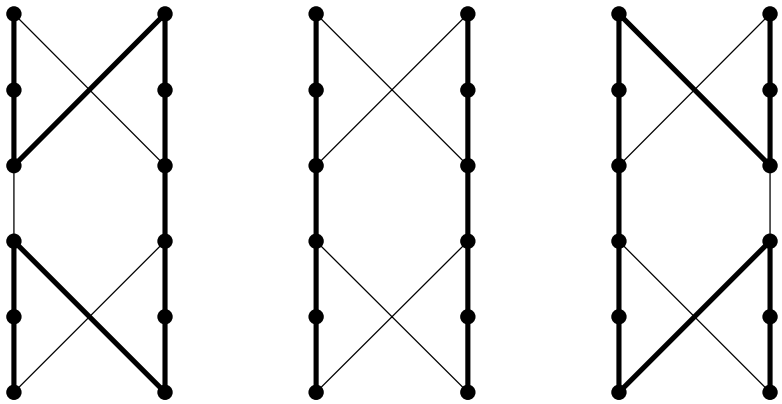


Fig. 2.5. Traversing a cover-testing component

Then $G' = (V', E')$ is the union of all these vertices and edges:

$$V' := \{a_1, \ldots, a_k\} \cup \bigcup_{e \in E} V'_e \qquad \text{and} \qquad E' := \bigcup_{e \in E} E'_e \cup \bigcup_{v \in V} E'_v \cup E''.$$

Obviously, G' can be constructed from G in polynomial time. Now suppose that G' contains a Hamiltonian cycle K. Let P be a trail contained in K beginning at a selector a_j and not containing any further selector. It is easy to see that P runs through exactly those T_e which correspond to all the edges incident with a certain vertex $v \in V$ (in the order given in Figure 2.6). Each of the T_e appears in one of the ways shown in Figure 2.5, and no vertices from other cover-testing components T_b (not corresponding to edges f incident with v) can occur. Thus the k selectors divide the Hamiltonian cycle K into k trails $P_1, \ldots, P_k$, each corresponding to a vertex $v \in V$. As K contains all the vertices of G' and as the vertices of an arbitrary cover-testing component T_f can only occur in K by occurring in a trail corresponding to one of the vertices incident with f, the k vertices of V determined by the trails $P_1, \ldots, P_k$ form a vertex cover W of G.

Conversely, let W be a vertex cover of G, where $|W| \leq k$. We may assume $|W| = k$ (because W remains a vertex cover if arbitrary vertices are added to it). Write $W = \{v_1, \ldots, v_k\}$. The edge set of the desired Hamiltonian cycle K is determined as follows. For each edge $e = uv$ of G we choose the thick edges in T_e drawn in one of the three graphs of Figure 2.5, where our choice depends on the intersection of W with e as follows:

- if $W \cap e = \{u\}$, we choose the edges of the graph on the left;
- if $W \cap e = \{v\}$, we choose the edges of the graph on the right;
- if $W \cap e = \{u, v\}$, we choose the edges of the graph in the middle.

Moreover, K contains all edges in E'_{v_i} (for $i = 1, \ldots, k$) and the edges

$$\begin{array}{ll} \{a_i, (v_i, (ev_i)_1, 1)\} & \text{for } i = 1, \ldots, k; \\ \{a_{i+1}, (v_i, (ev_i)_{\deg v_i}, 6)\} & \text{for } i = 1, \ldots, k-1; \quad \text{and} \\ \{a_1, (v_k, (ev_k)_{\deg v_k}, 6)\}. & \end{array}$$

The reader may check that K is indeed a Hamiltonian cycle for G'. □

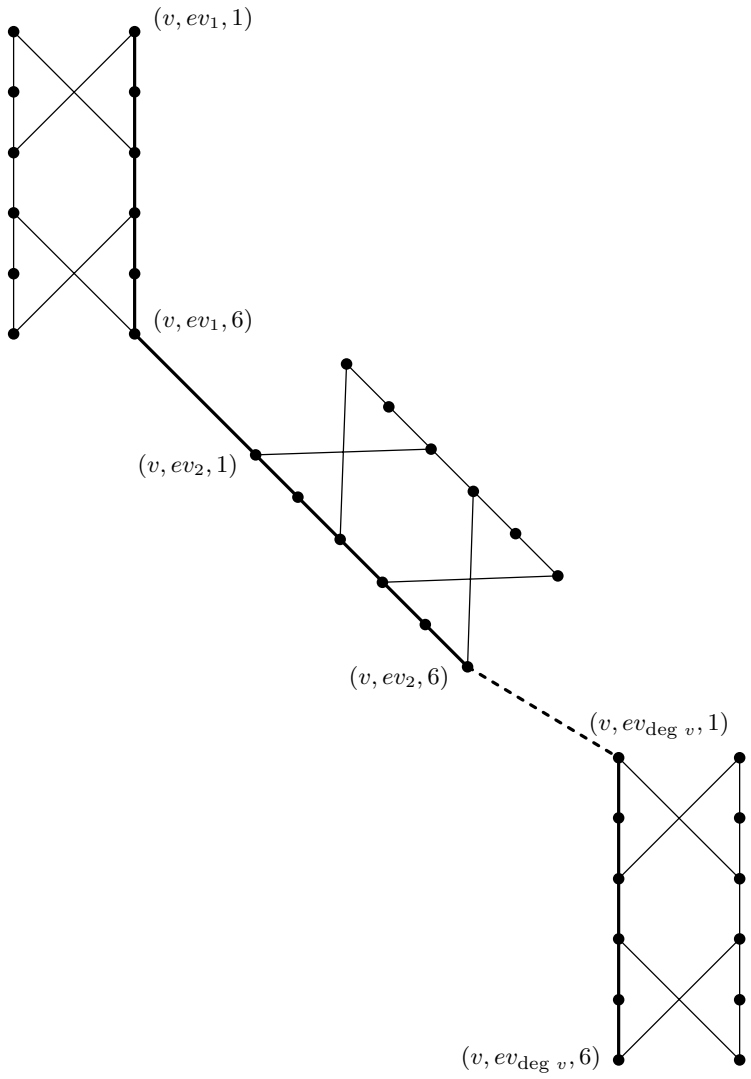


Fig. 2.6. The path associated with the vertex v

3

Shortest Paths

So many paths that wind and wind...
Ella Wheeler Wilcox

One of the most common applications of graphs in everyday life is representing networks for traffic or for data communication. The schematic map of the German motorway system in the official guide *Autobahn Service*, the railroad or bus lines in some public transportation system, and the network of routes an airline offers are routinely represented by graphs. Therefore it is obviously of great practical interest to study paths in such graphs. In particular, we often look for paths which are good or even best in some respect: sometimes the shortest or the fastest route is required, sometimes we want the cheapest path or the one which is safest – for example, we might want the route where we are least likely to encounter a speed-control installation. Thus we will study shortest paths in graphs and digraphs in this chapter; as we shall see, this is a topic whose interest extends beyond traffic networks.

3.1 Shortest paths

Let $G = (V, E)$ be a graph or a digraph on which a mapping $w : E \to \mathbb{R}$ is defined. We call the pair (G, w) a *network*; the number $w(e)$ is called the *length* of the edge e. Of course, this terminology is not intended to exclude other interpretations such as cost, duration, capacity, weight, or probability; we will encounter several examples later. For instance, in the context of studying spanning trees, we usually interpret $w(e)$ as the weight of the edge e. But in the present chapter the reader should keep the intuitive interpretation of distances in a network of streets in mind. This naturally leads to the following definition. For each walk $W = (e_1, \ldots, e_n)$, the *length* of W is

$$w(W) \;:=\; w(e_1) + \ldots + w(e_n),$$

where, of course, W has to be directed for digraphs. It is tempting to define the distance $d(a, b)$ between two vertices a and b in G as the minimum over all lengths of walks with start vertex a and end vertex b. However, there are

two difficulties with this definition: first, b might not be accessible from a, and second, a minimum might fail to exist. The first problem is solved by putting $d(a,b) = \infty$ if b is not accessible from a. The second problem arises from the possible existence of cycles of negative length. For example, in the network shown in Figure 3.1, we can find a walk of arbitrary negative length from a to b by using the cycle (x, y, z, x) as often as needed. In order to avoid this problem, one usually restricts attention to paths.

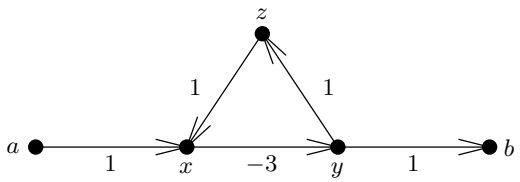


Fig. 3.1. A network

Thus we formally define the *distance* $d(a,b)$ between two vertices a and b in (G,w) as follows:

$$d(a,b) \;=\; \begin{cases} \infty & \text{if } b \text{ is not accessible from } a, \\ \min\{w(W)\colon W \text{ is a path from } a \text{ to } b\} & \text{otherwise.} \end{cases} \tag{3.1}$$

Most of the networks we will deal with will not contain any cycles of negative length; then the distance between two vertices is well-defined even if we would allow walks in the definition. Any path W (directed in the case of digraphs) for which the minimum in equation (3.1) is achieved is called a *shortest path* from a to b.

Note that always $d(a,a) = 0$, since an empty sum is considered to have value 0, as usual. If we talk of shortest paths and distances in a graph (or a digraph) without giving any explicit length function, we always use the length function which assigns length $w(e) = 1$ to each edge e.

The reader might wonder why negative lengths are allowed at all and whether they occur in practice. The answer is yes, they do occur, as the following example taken from [Law76] shows; this also provides a first example for another interpretation of the length of an edge.

Example 3.1.1. A trading ship travels from port a to port b, where the route (and possible intermediary ports) may be chosen freely. The routes are represented by trails in a digraph G, and the length $w(e)$ of an edge $e = xy$ signifies the profit gained by going from x to y. For some edges, the ship might have to travel empty so that $w(e)$ is negative for these edges: the profit is actually a loss. Replacing w by $-w$ in this network, the shortest path represents the route which yields the largest possible profit.

Clearly, the practical importance of the preceding example is negligible. We will encounter genuinely important applications later when treating flows

and circulations, where the existence of cycles of negative length – and finding such cycles – will be an essential tool for determining an optimal circulation.

We now give an example for an interpretation of shortest paths which allows us to formulate a problem (which at first glance might seem completely out of place here) as a problem of finding shortest paths in a suitable graph.

Example 3.1.2. In many applications, the length of an edge signifies the probability of its failing – for instance, in networks of telephone lines, or broadcasting systems, in computer networks, or in transportation routes. In all these cases, one is looking for the route having the highest probability for not failing. Let $p(i,j)$ be the probability that edge (i,j) does not fail. Under the – not always realistic – assumption that failings of edges occur independently of each other, $p(e_1)\dots p(e_n)$ gives the probability that the walk $(e_1,\dots,e_n)$ can be used without interruption. We want to maximize this probability over all possible walks with start vertex a and end vertex b. Note first that the maximum of the product of the $p(e)$ is reached if and only if the logarithm of the product, namely $\log p(e_1)+\dots+\log p(e_n)$, is maximal. Moreover, $\log p(e) \leq 0$ for all e, since $p(e) \leq 1$. We now put $w(e) = -\log p(e)$; then $w(e) \geq 0$ for all e, and we have to find a walk from a to b for which $w(e_1) + \dots + w(e_n)$ becomes minimal. Thus our problem is reduced to a shortest path problem. In particular, this technique solves the problem mentioned in our introductory remarks – finding a route where it is least likely that our speed will be controlled by the police – provided that we know for all edges the probability of a speed check.

In principle, any technique for finding shortest paths can also be used to find *longest paths*: replacing w by $-w$, a longest path with respect to w is just a shortest path with respect to $-w$. However, good algorithms for finding shortest paths are known only for the case where G does not contain any cycles of negative length. In the general case we basically have to look at all possible paths. Note that replacing w by $-w$ in general creates cycles of negative length.

Exercise 3.1.3 (knapsack problem). Consider n given objects, each of which has an associated *weight* a_j and also a *value* c_j, where both the a_j and the c_j are positive integers. We ask for a subset of these objects such that the sum of their weights does not exceed a certain bound b and such that the sum of their values is maximal. Packing a knapsack provides a good example, which explains the terminology used. Reduce this problem to finding a longest path in a suitable network. Hint: Use an acyclic network with a start vertex s, an end vertex t, and $b+1$ vertices for each object.

3.2 Finite metric spaces

Before looking at algorithms for finding shortest paths, we want to show that there is a connection between the notions of distance and metric space. We

recall that a *metric space* is a pair (X, d) consisting of a set X and a mapping $d\colon X^2 \to \mathbb{R}_0^+$ satisfying the following three conditions for all $x, y, z \in X$:

(MS1) $d(x, y) \geq 0$, and $d(x, y) = 0$ if and only if $x = y$;
(MS2) $d(x, y) = d(y, x)$;
(MS3) $d(x, z) \leq d(x, y) + d(y, z)$.

The value $d(x, y)$ is called the *distance* between x and y; the inequality in (MS3) is referred to as the *triangle inequality*. The matrix $D = (d(x, y))_{x,y \in X}$ is called the *distance matrix* of (X, d).

Now consider a network (G, w), where G is a graph and w is a positive valued mapping $w : E \to \mathbb{R}^+$. Note that a walk with start vertex a and end vertex b which has length $d(a, b)$ – where the distance between a and b is defined as in Section 3.1 – is necessarily a path. The following result states that our use of the term *distance* in this context is justified; the simple proof is left to the reader.

Lemma 3.2.1. *Let $G = (V, E)$ be a connected graph with a positive length function w. Then (V, d) is a finite metric space, where the distance function d is defined as in Section 3.1.* □

Lemma 3.2.1 suggests the question whether any finite metric space can be realized by a network. More precisely, let D be the distance matrix of a finite metric space (V, d). Does a graph $G = (V, E)$ with length function w exist such that its distance matrix with respect to w agrees with D? Hakimi and Yau [HaVa64] answered this question as follows.

Proposition 3.2.2. *Any finite metric space can be realized by a network with a positive length function.*

Proof. Let (V, d) be a finite metric space. Choose G to be the complete graph with vertex set V, and let the length function w be the given distance function d. By d' we denote the distance in the network (G, w) as defined in Section 3.1; we have to show $d = w = d'$. Thus let $W = (e_1, \ldots, e_n)$ be a trail with start vertex a and end vertex b. For $n \geq 2$, an iterative application of the triangle inequality yields:

$$w(W) = w(e_1) + \ldots + w(e_n) = d(e_1) + \ldots + d(e_n) \geq d(a, b).$$

As the one edge path a — b has length $d(a, b)$, we are finished. □

Exercise 3.2.3. Find a condition under which a finite metric space can be realized by a graph, that is, by a network all of whose edges have length 1; see [KaCh65].

We have only considered the case where a metric space (V, d) is realized by a network on the vertex set V. More generally, we could allow a network on a graph $G = (V', E)$ with $V \subset V'$, where the distance $d_G(a, b)$ in G for

two vertices a, b of V is the same as their distance $d(a, b)$ in the metric space. Such a realization is called *optimal* if the sum of all lengths of edges is minimal among all possible realizations. It is not obvious that such optimal realizations exist, but they do; see [Dre84] and [ImSZ84].

Example 3.2.4. The following simple example shows that the realization given in the proof of Proposition 3.2.2 is not necessarily optimal. Let $d(a, b) = d(b, c) = 4$ and $d(a, c) = 6$. Note that the realization on K_3 has total length 14. But there is also a realization on four vertices with total length just seven; see Figure 3.2.

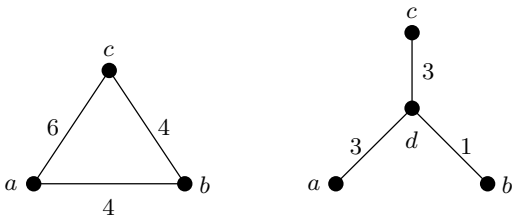


Fig. 3.2. Two realizations of a distance matrix

Realizations of metric spaces by networks have been intensively studied. In particular, the question whether a given metric space can be realized on a tree has sparked considerable interest; such a realization is necessarily optimal [HaVa64]. Bunemann [Bun74] proved that a realization on a tree is possible if and only if the following condition holds for any four vertices x, y, z, t of the given metric space:

$$d(x, y) + d(z, t) \leq \max\,(d(x, z) + d(y, t), d(x, t) + d(y, z)).$$

A different characterization (using ultra-metrics) is due to [Ban90]. We also refer the reader to [Sim88] and [Alt88]. The problem of finding an optimal realization is difficult in general: it is NP-hard [Win88].

3.3 Breadth first search and bipartite graphs

We now turn to examining algorithms for finding shortest paths. All techniques presented here also apply to multigraphs, but this generalization is of little interest: when looking for shortest paths, out of a set of parallel edges we only use the one having smallest length. In this section, we consider a particularly simple special case, namely distances in graphs (where each edge has length 1). The following algorithm was suggested by Moore [Moo59] and is known as *breadth first search*, or, for short, BFS. It is one of the most fundamental methods in algorithmic graph theory.

Algorithm 3.3.1 (BFS). Let G be a graph or digraph given by adjacency lists A_v. Moreover, let s be an arbitrary vertex of G and Q a queue.[1] The vertices of G are labelled with integers $d(v)$ as follows:

Procedure BFS$(G, s; d)$

```
(1) Q ← ∅; d(s) ← 0
(2) append s to Q
(3) while Q ≠ ∅ do
(4)       remove the first vertex v from Q;
(5)       for w ∈ A_v do
(6)             if d(w) is undefined
(7)             then d(w) ← d(v) + 1; append w to Q
(8)             fi
(9)       od
(10) od
```

Theorem 3.3.2. *Algorithm 3.3.1 has complexity $O(|E|)$. At the end of the algorithm, every vertex t of G satisfies*

$$d(s,t) = \begin{cases} d(t) & \text{if } d(t) \text{ is defined,} \\ \infty & \text{otherwise.} \end{cases}$$

Proof. Obviously, each edge is examined at most twice by BFS (in the directed case, only once), which yields the assertion about the complexity. Moreover, $d(s,t) = \infty$ if and only if t is not accessible from s, and thus $d(t)$ stays undefined throughout the algorithm. Now let t be a vertex such that $d(s,t) \neq \infty$. Then $d(s,t) \leq d(t)$, since t was reached by a path of length $d(t)$ from s. We show that equality holds by using induction on $d(s,t)$. This is trivial for $d(s,t) = 0$, that is, $s = t$. Now assume $d(s,t) = n+1$ and let $(s, v_1, \ldots, v_n, t)$ be a shortest path from s to t. Then $(s, v_1, \ldots, v_n)$ is a shortest path from s to v_n and, by our induction hypothesis, $d(s, v_n) = n = d(v_n)$. Therefore $d(v_n) < d(t)$, and thus BFS deals with v_n before t during the **while**-loop. On the other hand, G contains the edge $v_n t$ so that BFS certainly reaches t when examining the adjacency list of v_n (if not earlier). This shows $d(t) \leq n+1$ and hence $d(t) = n+1$. □

Corollary 3.3.3. *Let s be a vertex of a graph G. Then G is connected if and only if $d(t)$ is defined for each vertex t at the end of* BFS$(G, s; d)$. □

Note that the statement analogous to Corollary 3.3.3 for directed graphs is not true. If we want to check whether a given digraph is connected, we should apply BFS to the corresponding graph $|G|$. Applying BFS$(G, s; d)$ for each

[1] Recall that a *queue* is a data structure for which elements are always appended at the end, but removed at the beginning (*first in – first out*). For a discussion of the implementation of queues we refer to [AhHU83] or [CoLR90].

vertex s of a digraph allows us to decide whether G is strongly connected; clearly, this holds if and only if BFS$(G, s; d)$ always reaches all vertices t and assigns values to $d(t)$. However, this method is not very efficient, as it has complexity $O(|V||E|)$. In Chapter 8, we will see a much better technique which has complexity $O(|E|)$.

For an example, let us consider how BFS runs on the digraph G drawn in Figure 3.3. To make the algorithm deterministic, we select the vertices in alphabetical order in step (5) of the BFS. In Figures 3.4 and 3.5, we illustrate the output of BFS both for G and the associated graph $|G|$. To make things clearer, we have drawn the vertices in *levels* according to their distance to s; also, we have omitted all edges leading to vertices already labelled. Thus all we see of $|G|$ is a *spanning tree*, that is, a spanning subgraph of G which is a tree. This kind of tree will be studied more closely in Chapter 4. Note that distances in G and in $|G|$ do not always coincide, as was to be expected. However, we always have $d_G(s,t) \geq d_{|G|}(s,t)$.

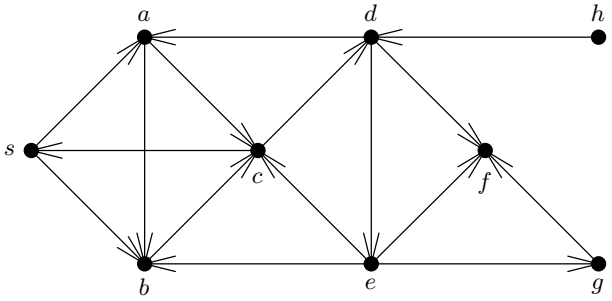


Fig. 3.3. A digraph G

Exercise 3.3.4. Design a BFS-based algorithm COMP(G) which determines the connected components of a graph G.

Next we consider a particularly important class of graphs, namely the bipartite graphs. As we shall see soon, BFS gives an easy way to decide whether or not a graph belongs to this class. Here a graph $G = (V, E)$ is said to be *bipartite* if there is a partition $V = S \dot\cup T$ of its vertex set such that the sets of edges $E|S$ and $E|T$ are empty, that is, each edge of G is incident with one vertex in S and one vertex in T. The following theorem gives a very useful characterization of these graphs.

Theorem 3.3.5. *A graph G is bipartite if and only if it does not contain any cycles of odd length.*

Proof. First suppose that G is bipartite and let $V = S \dot\cup T$ be the corresponding partition of its vertex set. Consider an arbitrary closed trail in G,

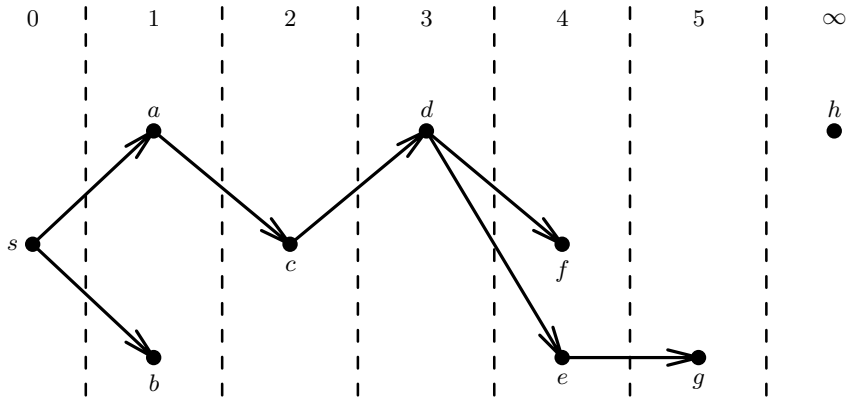


Fig. 3.4. BFS-tree for G

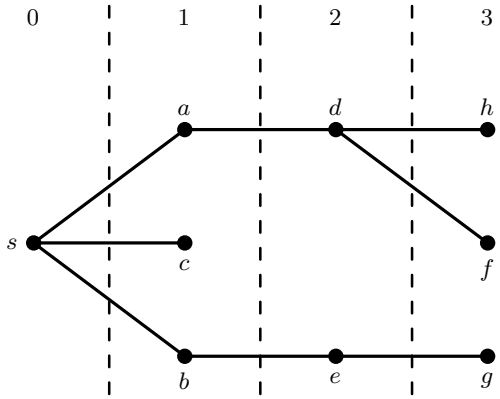


Fig. 3.5. BFS-tree for $|G|$

say

$$C: \; v_1 \;\text{—}\; v_2 \;\text{—}\; \ldots \;\text{—}\; v_n \;\text{—}\; v_1.$$

We may assume $v_1 \in S$. Then

$$v_2 \in T, \quad v_3 \in S, \quad v_4 \in T, \quad \ldots \quad , \quad v_n \in T, \quad v_1 \in S,$$

as there are no edges within S or T. Hence n must be even.

Conversely, suppose that G does not contain any cycles of odd length. We may assume that G is connected. Choose some vertex x_0. Let S be the set of all vertices x having even distance $d(x, x_0)$ from x_0, and let T be the complement of S. Now suppose that there is an edge xy in G with $x, y \in S$.

Let W_x and W_y be shortest paths from x_0 to x and y, respectively. By our definition of S, both these paths have even length. Let us denote the last common vertex of W_x and W_y by z (traversing both paths starting at x_0), and call their final parts (leading from z to x and y, respectively) W'_x and W'_y. Then it is easily seen that

$$x \quad \overline{W'_x} \quad z \quad \overline{W'_y} \quad y \quad \overline{xy} \quad x$$

is a cycle of odd length in G, a contradiction. Similarly, G cannot contain an edge xy with $x, y \in T$. Hence $S \dot\cup T$ is a partition of V such that there are no edges within S or T, and G is bipartite. □

The proof of Theorem 3.3.5 shows how we may use the distances $d(s,t)$ in G (from a given start vertex s) for finding an appropriate partition of the vertex set of a given bipartite graph G. These distances can be determined using BFS; of course, we should modify Algorithm 3.3.1 in such a way that it detects cycles of odd length, in case G is not bipartite. This is actually rather simple: when BFS examines an edge e for the first time, a cycle of odd length containing e exists if and only if e has both its vertices in the same level. This gives us the desired criterion for checking whether G is bipartite or not; moreover, if G is bipartite, the part of G determined by s consists of those vertices which have even distance from s. These observations lead to the following algorithm and the subsequent theorem.

Algorithm 3.3.6. Let G be a connected graph and s a vertex of G.

Procedure BIPART($G, s; S, T$,bip)

```
(1) Q ← ∅, d(s) ← 0, bip ← true, S ← ∅;
(2) append s to Q;
(3) while Q ≠ ∅ and bip = true do
(4)       remove the first vertex v of Q;
(5)       for w ∈ A_v do
(6)             if d(w) is undefined
(7)             then d(w) ← d(v) + 1; append w to Q
(8)             else if d(v) = d(w) then bip ← false fi
(9)             fi
(10)      od
(11) od
(12) if bip = true
(13) then for v ∈ V do
(14)            if d(v) ≡ 0 (mod 2) then S ← S ∪ {v} fi
(15)      od
(16)      T ← V \ S
(17) fi
```

Theorem 3.3.7. *Algorithm 3.3.6 checks with complexity $O(|E|)$ whether a given connected graph G is bipartite; if this is the case, it also determines the corresponding partition of the vertex set.* □

Exercise 3.3.8. Describe a BFS-based algorithm which finds with complexity $O(|V||E|)$ a shortest cycle in – and thus the girth of – a given graph G.

The problem of finding a shortest cycle was extensively studied by Itai and Rodeh [ItRo78] who also treated the analogous problem for directed graphs. The best known algorithm has a complexity of $O(|V|^2)$; see [YuZw97]. BFS can also be used to find a shortest cycle of even or odd length, respectively; see [Mon83].

3.4 Shortest path trees

We now turn to the problem of determining shortest paths in a general network; actually, all known algorithms for this problem even find a shortest path from the start vertex s to each vertex t which is accessible from s. Choosing t in a special way does not decrease the (worst case) complexity of the algorithms.

In view of the discussion in Section 3.1, we will always assume that the given network (G, w) does not contain any cycles of negative length, so that the length of a shortest walk between two specified vertices always equals their distance. Moreover, we assume from now on that G is a directed graph so that all paths involved also have to be directed.[2] We will always tacitly use these assumptions, even if they are not stated explicitly.

In the present section, we will not yet provide any algorithms but deal with some theoretical results first. We start with a simple but fundamental observation.

Lemma 3.4.1. *Let W be a shortest path from s to t in the network (G, w), and let v be a vertex on W. Then the subpath W from s to v is a shortest path from s to v.*

Proof. Denote the subpaths of W from s to v and from v to t by W_1 and W_2, respectively. Let us assume the existence of a path W_1' from s to v which is shorter than W_1. Then $W' = W_1'W_2$ is a walk from s to t which is shorter than W. But W' contains a path W'' from s to t, which may be constructed from W' by removing cycles; see Exercise 1.6.6. By our general hypothesis, (G, w) does not contain any cycles of negative length, and hence W'' has to be shorter than W, a contradiction. □

[2] For nonnegative length functions, the undirected case can be treated by considering the complete orientation $\vec{G}$ instead of G. If we want to allow edges of negative length, we need a construction which is considerably more involved, see Section 14.6.

Our next result characterizes the shortest paths with start vertex s in terms of the distances $d(s,v)$:

Theorem 3.4.2. *Let P be a path from s to t in G. Then P is a shortest path if and only if each edge $uv \in P$ satisfies the condition*

$$d(s,v) \;=\; d(s,u) + w(uv). \tag{3.2}$$

Proof. First, let P be a shortest path from s to t. By Lemma 3.4.1, the subpaths of P from s to u and from s to v are likewise shortest paths. Thus P has to satisfy condition (3.2).

Conversely, let P be a path satisfying condition (3.2), say $P = (v_0, \ldots, v_k)$, where $v_0 = s$ and $v_k = t$. Then

$$\begin{aligned} d(s,t) &= d(s,v_k) - d(s,v_0) \\ &= \big(d(s,v_k) - d(s,v_{k-1})\big) + \big(d(s,v_{k-1}) - d(s,v_{k-2})\big) \\ &\qquad + \ldots + \big(d(s,v_1) - d(s,v_0)\big) \\ &= w(v_{k-1}v_k) + w(v_{k-2}v_{k-1}) + \ldots + w(v_0v_1) \\ &= w(P), \end{aligned}$$

so that P is indeed a shortest path from s to t. □

Theorem 3.4.2 leads to a very efficient way of storing a system of shortest paths with start vertex s. We need a definition: a *spanning arborescence* with root s (that is, of course, a spanning subdigraph which is a directed tree with root s) is called a *shortest path tree* for the network (G,w) if, for each vertex v, the unique path from s to v in T has length $d(s,v)$; we will often use the shorter term *SP-tree*. The following characterization of SP-trees is an immediate consequence of Theorem 3.4.2.

Corollary 3.4.3. *Let T be a spanning arborescence of G with root s. Then T is a shortest path tree if and only if each edge $uv \in T$ satisfies condition (3.2).* □

In Exercise 3.4.5, we will provide an alternative characterization of SP-trees which does not involve the distances in the network (G,w). It remains to prove the existence of SP-trees:

Theorem 3.4.4. *Let G be a digraph with root s, and let $w\colon E \to \mathbb{R}$ be a length function on G. If the network (G,w) does not contain any directed cycles of negative length, then there exists an SP-tree with root s for (G,w).*

Proof. As s is a root for G, we can reach any other vertex v of G via a shortest path, say P_v. Let E' be the union of the edge sets of all these paths. Then the digraph $G' = (V,E')$ still has s as a root, and hence contains a spanning arborescence with root s, say T. By Theorem 3.4.2, all edges of all the paths P_v satisfy condition (3.2). Hence all edges in E' – and, in particular, all edges of T – satisfy that condition. By Corollary 3.4.3, T has to be an SP-tree for (G,w). □

Exercise 3.4.5. Let (G, w) be a network which does not contain any directed cycles of negative length, let s be a root of G, and assume that the distances $d(s, v)$ are already known. Outline an algorithm capable of computing with complexity $O(|E|)$ an SP-tree.

Exercise 3.4.6. Let T be a spanning arborescence with root s in a network (G, w) which does not contain any directed cycles of negative length. Show that T is an SP-tree if and only if the following condition holds for each edge $e = uv$ of G:

$$d_T(s, v) \;\leq\; d_T(s, u) + w(uv), \tag{3.3}$$

where $d_T(s, u)$ denotes the distance from s to u in the network $(T, w|T)$.

Exercise 3.4.7. Show by example that the condition that no cycles of negative length exist is necessary for proving Theorem 3.4.4: if this condition is violated, then (G, w) may not admit any SP-trees. Also give an example where an SP-tree exists, even though (G, w) contains a directed cycle of negative length.

3.5 Bellman's equations and acyclic networks

In this section, we first continue our theoretical discussion by deriving a system of equations which has to be satisfied by the distances $d(s, v)$ from a specified start vertex s. For acyclic networks, these equations will lead to a linear time algorithm for actually computing the distances.

Again, we assume that the given network (G, w) does not contain any cycles of negative length. Without loss of generality, we let G have vertex set $V = \{1, \ldots, n\}$. Let us write $w_{ij} := w(ij)$ if G contains the edge ij, and $w_{ij} = \infty$ otherwise. Furthermore, let u_i denote the distance $d(s, i)$, where s is the start vertex; in most cases, we will simply take $s = 1$.

Now any shortest path from s to i has to contain a final edge ki, and deleting this edge yields a shortest path from s to k, by Lemma 3.4.1. Hence the distances u_i have to satisfy the following system of equations due to Bellman [Bel58], where we assume for the sake of simplicity that $s = 1$.

Proposition 3.5.1 (Bellman's equations). *Let $s = 1$. Then*

$$\text{(B)} \quad u_1 = 0 \quad \textit{and} \quad u_i = \min\{u_k + w_{ki} : i \neq k\} \quad \textit{for } i = 2, \ldots, n. \quad \square$$

As usual, we shall assume that each vertex is accessible from $s = 1$. Unfortunately, our standard hypothesis – namely, that (G, w) does not contain any cycles of negative length – is not strong enough to ensure that the distances $d(s, j)$ are the *only* solution to (B):

Exercise 3.5.2. Give an example of a network which contains no cycles of negative length but cycles of length 0, such that the system of equations (B) has at least two distinct solutions. Try to find a general construction providing infinitely many examples for this phenomenon.

We will now prove that the system of equations (B) does have a unique solution – namely the distances $d(1,j)$ – under the stronger assumption that (G,w) contains only cycles of positive length. To this purpose, we first construct a spanning arborescence T with root 1 satisfying the condition

$$d_T(1,i) \;=\; u_i \quad \text{for } i = 1,\dots,n \tag{3.4}$$

from any given solution $(u_1,\dots,u_n)$ of (B). Let us choose some vertex $j \neq 1$. We want to construct a path of length u_j from 1 to j. To do so, we first choose some edge ij with $u_j = u_i + w_{ij}$, then an edge hi with $u_i = u_h + w_{hi}$, etc. Let us show that this construction cannot yield a cycle. Suppose, to the contrary, we were to get a cycle, say

$$C:\; v_1 \;\text{—}\; v_2 \;\text{—}\; \dots \;\text{—}\; v_k \;\text{—}\; v_1.$$

Then we would have the following equations which imply $w(C) = 0$, a contradiction to our assumption that G contains cycles of positive length only:

$$\begin{aligned} u_{v_1} &= u_{v_k} + w_{v_k v_1} \\ &= u_{v_{k-1}} + w_{v_{k-1} v_k} + w_{v_k v_1} \\ &= \dots \;=\; u_{v_1} + w_{v_1 v_2} + \dots + w_{v_k v_1} \\ &= u_{v_1} + w(C). \end{aligned}$$

Thus our construction can only stop at the special vertex 1, yielding a path from 1 to j. A similar computation shows that, for each vertex i occurring on this path, the part of the path leading to i has length u_i. Continuing in this way for all other vertices not yet occurring in the path(s) constructed so far – where we construct a new path backward only until we reach some vertex on one of the paths constructed earlier – we obtain a spanning arborescence with root 1 satisfying equation (3.4), say T.[3] Let us check that T is actually an SP-tree, so that the u_j are indeed the distances $d(1,j)$. It suffices to verify the criterion of Exercise 3.4.6. But this is easy: consider any vertex $i \neq 1$. As the u_j solve (B) and as (3.4) holds,

$$d_T(s,i) \;=\; u_i \;\leq\; u_k + w_{ki} \;=\; d_T(s,k) + w(ki)$$

for every edge ki of G. This already proves the desired uniqueness result:

Theorem 3.5.3. *If 1 is a root of G and if all cycles of G have positive length with respect to w, then Bellman's equations have a unique solution, namely the distances $u_j = d(1,j)$.* □

[3] In particular, we may apply this technique to the distances in G, since they satisfy Bellman's equations, which again proves the existence of SP-trees (under the stronger assumption that (G,w) contains only cycles of positive length).

In view of the preceding results, we would now like to *solve* the system of equations (B). We begin with the simplest possible case, where G is an acyclic digraph. As we saw in Section 2.6, we can find a topological sorting of G in $O(|E|)$ steps. After having executed TOPSORT, let us replace each vertex v by its number topnr(v). Then every edge ij in G satisfies $i < j$, and we may simplify Bellman's equations as follows:

$$u_1 = 0 \quad \text{and} \quad u_i = \min\{u_k + w_{ki} : k = 1, \ldots, i-1\} \quad \text{for } i = 2, \ldots, n.$$

Obviously, this system of equations can be solved recursively in $O(|E|)$ steps if we use backward adjacency lists, where each list contains the edges with a specified head. This proves the following result.

Theorem 3.5.4. *Let N be a network on an acyclic digraph G with root s. Then one can determine the distances from s (and hence a shortest path tree with root s) in $O(|E|)$ steps.* □

Mehlhorn and Schmidt [MeSc86] found a larger class of graphs (containing the acyclic digraphs) for which with complexity $O(|E|)$ it is possible to determine the distances with respect to a given vertex.

Exercise 3.5.5. Show that, under the same conditions as in Theorem 3.5.4, we can also with complexity $O(|E|)$ determine a system of *longest* paths from s to all other vertices. Does this yield an efficient algorithm for the knapsack problem of Exercise 3.1.3? What happens if we drop the condition that the graph should be acyclic?

3.6 An application: Scheduling projects

We saw in Exercise 3.5.5 that it is easy to find longest paths in an acyclic digraph. We will use this fact to solve a rather simple instance of the problem of making up a schedule for a project. If we want to carry out a complex project – such as, for example, building a dam, a shopping center or an airplane – the various tasks ought to be well coordinated to avoid loss of time and money. This is the goal of network planning, which is, according to [Mue73] 'the tool from operations research used most.' [Ta92] states that these techniques 'enjoy tremendous popularity among practitioners in the field'. We restrict ourselves to the simple case where we have restrictions on the chronological sequence of the tasks only: there are some tasks which we cannot begin before certain others are finished. We are interested in the shortest possible time the project takes, and would like to know the points of time when each of the tasks should be started. Two very similar methods to solve this problem, namely the *critical path method (CPM)* and the *project evaluation and review technique (PERT)* were developed between 1956 and 1958 by two different groups, cf. [Ta92] and [Mue73]. CPM was introduced by E. I. du Pont de Nemours &

Company to help schedule construction projects, and PERT was developed by Remington Rand for the U.S. Navy to help schedule the research and development activities for the Polaris missile program. CPM-PERT is based on determining longest paths in an acyclic digraph. We shall use a formulation where the activities in the project are represented by vertices; alternatively, one could also represent them by arcs, cf. [Ta92].

First, we assign a vertex $i \in \{1, \dots, N\}$ of a digraph G to each of the N tasks of our project. We let ij be an edge of G if and only if task i has to be finished before beginning task j. The edge ij then has length $w_{ij} = d_i$ equal to the time task i takes. Note that G has to be acyclic, because otherwise the tasks in a cycle in G could never be started. As we have seen in Lemma 2.6.2, G contains at least one vertex v with $d_{\text{in}}(v) = 0$ and, analogously, at least one vertex w with $d_{\text{out}}(w) = 0$. We introduce a new vertex s (the start of the project) and add edges sv for all vertices v with $d_{\text{in}}(v) = 0$; similarly, we introduce a new vertex z (the end of the project) and add edges wz for all vertices w with $d_{\text{out}}(w) = 0$. All the new edges sv have length 0, whereas the edges wz are given length d_w. In this way we get a larger digraph H with root s; by Theorem 2.6.3, we may assume H to be topologically sorted.

Now we denote the earliest possible point of time at which we could start task i by t_i. As all the tasks immediately preceding i have to be finished before, we get the following system of equations:

$$\text{(CPM)} \qquad t_s = 0 \quad \text{and} \quad t_i = \max\{t_k + w_{ki} : ki \text{ an edge in } H\}.$$

This system of equations is analogous to Bellman's equations and describes the longest paths in H, compare Exercise 3.5.5. As in Theorem 3.5.3, (CPM) has a unique solution which again is easy to calculate recursively, since H is topologically sorted and thus only contains edges ij with $i < j$. The minimal amount of time the project takes is the length $T = t_z$ of a longest path from s to z. If the project is actually to be finished at time T, the latest point of time T_i where we can still start task i is given recursively by

$$T_z = T \qquad \text{and} \qquad T_i = \min\{T_j - w_{ij} : ij \text{ an edge in } H\}.$$

Thus $T_z - T_i$ is the length of a longest path from i to z. Of course, we should get $T_s = 0$, which is useful for checking our calculations. The difference $m_i = T_i - t_i$ between the earliest point of time and the latest point of time for beginning task i is called *float* or *slack*. All tasks i having float $m_i = 0$ are called *critical*, because they have to be started exactly at the point of time $T_i = t_i$, as otherwise the whole project would be delayed. Note that each longest path from s to z contains critical tasks only; for that reason each such path is called a *critical path* for H. In general, there will be more than one critical path.

In practice, H will not contain all edges ij for which i has to be finished before j, but only those edges for which i is an immediate predecessor of j so that there are no intermediate tasks between i and j.

Table 3.1. Project of building a house

Vertex	Task	Amount of time	Preceding tasks
1	prepare the building site	3	–
2	deliver the building material	2	–
3	dig the foundation-hole	2	1,2
4	build the foundation	2	3
5	build the walls	7	4
6	build the roof supports	3	5
7	cover the roof	1	6
8	connect the water pipes to the house	3	4
9	plasterwork outside	2	7,8
10	install the windows	1	7,8
11	put in the ceilings	3	5
12	lay out the garden	4	9,10
13	install inside plumbing	5	11
14	put insulation on the walls	3	10,13
15	paint the walls	3	14
16	move	5	15

Example 3.6.1. As an example, let us consider a simplified schedule for building a house. First, we need a list of the tasks, the amount of time they take, and which tasks have to be finished before which other tasks; this information can be found in Table 3.1. The corresponding digraph is shown in Figure 3.6. We have drawn the edges as undirected edges to make the figure somewhat simpler: all edges are to be considered as directed from left to right.

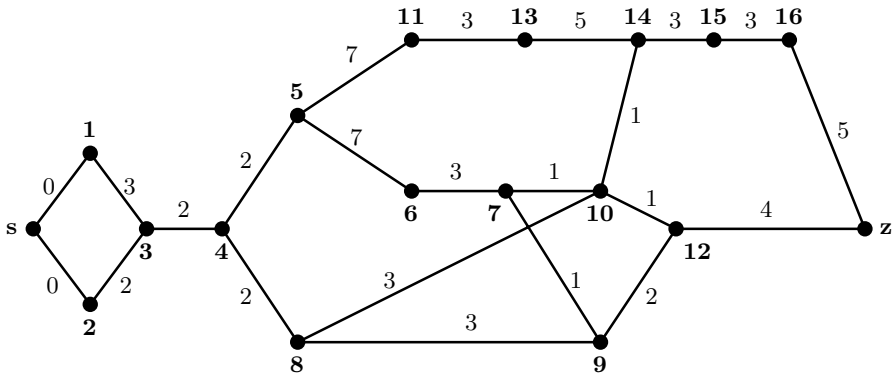


Fig. 3.6. Digraph for the project of building a house

The way the digraph is drawn in Figure 3.6, it is not necessary to state a topological sorting of the vertices explicitly; see Exercise 3.6.3. Using (CPM), we calculate consecutively

$$t_s = 0, \quad t_1 = 0, \quad t_2 = 0, \quad t_3 = 3, \quad t_4 = 5, \quad t_5 = 7, \quad t_8 = 7,$$
$$t_6 = 14, \quad t_{11} = 14, \quad t_{13} = 17, \quad t_7 = 17, \quad t_9 = 18, \quad t_{10} = 18,$$
$$t_{12} = 20, \quad t_{14} = 22, \quad t_{15} = 25, \quad t_{16} = 28, \quad T = t_z = 33.$$

Similarly, we compute the T_i and the floats m_i:

$$T_z = 33, \quad m_z = 0; \quad T_{16} = 28, \quad m_{16} = 0; \quad T_{15} = 25, \quad m_{15} = 0;$$
$$T_{12} = 29, \quad m_{12} = 9; \quad T_{14} = 22, \quad m_{14} = 0; \quad T_9 = 27, \quad m_9 = 9;$$
$$T_{10} = 21, \quad m_{10} = 3; \quad T_7 = 20, \quad m_7 = 3; \quad T_{13} = 17, \quad m_{13} = 0;$$
$$T_6 = 17, \quad m_6 = 3; \quad T_{11} = 14, \quad m_{11} = 0; \quad T_5 = 7, \quad m_5 = 0;$$
$$T_8 = 18, \quad m_8 = 11; \quad T_4 = 5, \quad m_4 = 0; \quad T_3 = 3, \quad m_3 = 0;$$
$$T_1 = 0, \quad m_1 = 0; \quad T_2 = 1, \quad m_2 = 1; \quad T_s = 0, \quad m_s = 0.$$

Thus the critical tasks are $s, 1, 3, 4, 5, 11, 13, 14, 15, 16, z$, and they form (in this order) the critical path, which is unique for this example.

Further information on project scheduling can be found in the books [Ta92] and [Mue73], and in the references given there. Of course, there is much more to scheduling than the simple method we considered. In practice there are often further constraints that have to be satisfied, such as scarce resources like limited amounts of machinery or restricted numbers of workers at a given point of time. For a good general overview of scheduling, the reader is referred to [LaLRS93]. We close this section with a couple of exercises; the first of these is taken from [Mue73].

Table 3.2. Project of building a new production facility

Vertex	Task	Amount of time	Preceding tasks
1	ask for offers, compare and order	25	–
2	take apart the old facility	8	–
3	remove the old foundation	5	2
4	plan the new foundation	9	1
5	term of delivery for the new facility	21	1
6	build the new foundation	9	3,4
7	install the new facility	6	5,6
8	train the staff	15	1
9	install electrical connections	2	7
10	test run	1	8,9
11	acceptance test and celebration	2	10

Exercise 3.6.2. A factory wants to replace an old production facility by a new one; the necessary tasks are listed in Table 3.2. Draw the corresponding network and determine the values t_i, T_i, and m_i.

Exercise 3.6.3. Let G be an acyclic digraph with root s. The *rank* $r(v)$ of a vertex v is the maximal length of a directed path from s to v. Use the methods introduced in this chapter to find an algorithm which determines the rank function.

Exercise 3.6.4. Let G be an acyclic digraph with root s, given by adjacency lists A_v. Show that the following algorithm computes the rank function on G, and determine its complexity:

Procedure RANK$(G, s; r)$

```
(1) create a list S_0, whose only element is s;
(2) r(s) ← 0; k ← 0;
(3) for  v ∈ V do d(v) ← d_in(v) od
(4) while S_k ≠ ∅ do
(5)       create a new list S_{k+1};
(6)       for v ∈ S_k do
(7)           for w ∈ A_v do
(8)               if d(w) = 1
(9)               then append w to S_{k+1}; r(w) ← k + 1; p(w) ← v
(10)              fi
(11)              d(w) ← d(w) − 1
(12)          od
(13)      od
(14)      k ← k + 1
(15) od
```

How can we determine $d(w)$? How can a longest path from s to v be found? Can RANK be used to find a topological sorting of G?

3.7 The algorithm of Dijkstra

In this section, we consider networks having all lengths nonnegative. In this case Bellman's equations can be solved by the algorithm of Dijkstra [Dij59], which is probably the most popular algorithm for finding shortest paths.

Algorithm 3.7.1. Let (G, w) be a network, where G is a graph or a digraph and all lengths $w(e)$ are nonnegative. The adjacency list of a vertex v is denoted by A_v. We want to calculate the distances with respect to a vertex s.

Procedure DIJKSTRA$(G, w, s; d)$

```
(1) d(s) ← 0, T ← V;
(2) for v ∈ V \ {s} do d(v) ← ∞ od
```

(3) **while** $T \neq \emptyset$ **do**
(4) find some $u \in T$ such that $d(u)$ is minimal;
(5) $T \leftarrow T \setminus \{u\}$;
(6) **for** $v \in T \cap A_u$ **do** $d(v) \leftarrow \min(d(v), d(u) + w_{uv})$ **od**
(7) **od**

Theorem 3.7.2. *Algorithm 3.7.1 determines with complexity $O(|V|^2)$ the distances with respect to some vertex s in (G, w). More precisely, at the end of the algorithm*

$$d(s,t) = d(t) \quad \textit{for each vertex } t.$$

Proof. Obviously, $d(t) = \infty$ if and only if t is not accessible from s. Now assume $d(t) \neq \infty$. Then $d(s,t) \leq d(t)$, as the algorithm reaches t via a directed path of length $d(t)$ from s to t. We will show the converse inequality $d(t) \leq d(s,t)$ by using induction on the order in which vertices are removed from T. The first vertex removed is s; trivially $d(s) = 0 = d(s,s)$. Now suppose that the inequality is true for all vertices t that were removed from T before u. We may assume that $d(u)$ is finite. Moreover, let

$$s = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \dots \xrightarrow{e_n} v_n = u$$

be a shortest path from s to u. Then

$$d(s, v_h) \;=\; \sum_{j=1}^{h} w(e_j) \qquad \text{for } h = 0, \dots, n.$$

Choose i as the maximal index such that v_i was removed from T before u. By the induction hypothesis,

$$d(s, v_i) \;=\; d(v_i) \;=\; \sum_{j=1}^{i} w(e_j).$$

Let us consider the iteration where v_i is removed from T in the **while** loop. As v_{i+1} is adjacent to v_i, the inequality $d(v_{i+1}) \leq d(v_i) + w(e_{i+1})$ is established during this iteration. But $d(v_{i+1})$ cannot be increased again in the subsequent iterations and, hence, this inequality still holds when u is removed. Thus

$$d(v_{i+1}) \;\leq\; d(v_i) + w(e_{i+1}) \;=\; d(s, v_i) + w(e_{i+1}) \;=\; d(s, v_{i+1}) \leq d(s, u). \quad (3.5)$$

Suppose first $v_{i+1} \neq u$, that is, $i \neq n-1$. By equation (3.5), $d(s,u) < d(u)$ would imply $d(v_{i+1}) < d(u)$; but then v_{i+1} would have been removed from T before u in view of the selection rule in step (4), contradicting the fact that we chose i to be maximal. Hence indeed $d(u) \leq d(s,u)$, as asserted. Finally, for $u = v_{i+1}$, the desired inequality follows directly from equation (3.5). This establishes the correctness of Dijkstra's algorithm. For the complexity, note

that in step (4) the minimum of the $d(v)$ has to be calculated (for $v \in T$), which can be done with $|T|-1$ comparisons. In the beginning of the algorithm, $|T| = |V|$, and then $|T|$ is decreased by 1 with each iteration. Thus we need $O(|V|^2)$ steps altogether for the execution of (4). It is easy to see that all other operations can also be done in $O(|V|^2)$ steps. □

We remark that the algorithm of Dijkstra might not work if there are negative weights in the network, even if no cycles of negative length exist. Note that the estimate in equation (3.5) does not hold any more if $w(e_{i+1}) < 0$. An algorithm which works also for negative weights can be found in Exercise 3.7.9.

Exercise 3.7.3. Modify Dijkstra's algorithm in such a way that it actually gives a shortest path from s to t, not just the distance $d(s,t)$. If s is a root of G, construct an SP-tree for (G,w).

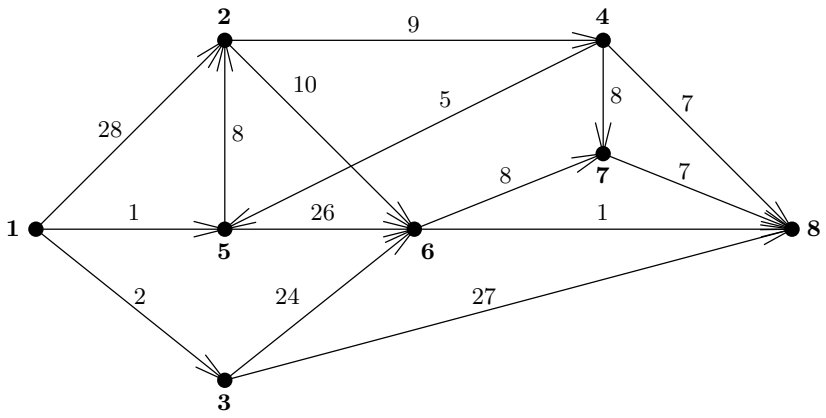


Fig. 3.7. A network

Example 3.7.4. Consider the network given in Figure 3.7 with vertex set $V = \{1,\dots,8\}$. With $s = 1$, Algorithm 3.7.1 is executed as follows, where the final values for d is indicated in bold face:

start values: $\mathbf{d(1) = 0}$, $d(i) = \infty$ for $i = 2,\dots,8$, $T = V$.
Iteration I: $u = 1$, $T = \{2,\dots,8\}$, $d(2) = 28$, $d(3) = 2$, $\mathbf{d(5) = 1}$;
Iteration II: $u = 5$, $T = \{2,3,4,6,7,8\}$, $d(2) = 9$, $\mathbf{d(3) = 2}$, $d(6) = 27$;
Iteration III: $u = 3$, $T = \{2,4,6,7,8\}$, $\mathbf{d(2) = 9}$, $d(6) = 26$, $d(8) = 29$;
Iteration IV: $u = 2$, $T = \{4,6,7,8\}$, $\mathbf{d(4) = 18}$, $d(6) = 19$;
Iteration V: $u = 4$, $T = \{6,7,8\}$, $\mathbf{d(6) = 19}$, $d(7) = 26$, $d(8) = 25$;
Iteration VI: $u = 6$, $T = \{7,8\}$, $\mathbf{d(8) = 20}$;
Iteration VII: $u = 8$, $T = \{7\}$, $\mathbf{d(7) = 26}$;
Iteration VIII: $u = 7$, $T = \emptyset$.

Exercise 3.7.5. Calculate the distances with respect to $s = 1$ for the underlying undirected network.

Let us return to the complexity of Dijkstra's algorithm. Initializing the variables in (1) and (2) takes $O(|V|)$ steps. During the entire **while** loop, each edge $e = uv$ is considered exactly once, namely during the iteration where u is removed from T. Thus step (6) contributes only $O(|E|)$ to the complexity of the algorithm, which is – at least for sparse graphs – much better than $O(|V|^2)$. Therefore it makes sense to try to reduce the number of comparisons in step (4) by using an appropriate data structure.

Recall that a *priority queue* (sometimes also called a *heap*) is a data structure consisting of a number of elements each of which is associated with a real number, its *priority*. Permissible operations include inserting elements according to their priority as well as determining and removing the element with the smallest priority; the latter operation is usually referred to as DELETEMIN. As shown in computer science, a priority queue with n elements can be implemented in such a way that each of these two operations can be executed with complexity $O(\log n)$; we need a refinement of this standard implementation which enables us also to remove a given element or reduce its priority with the same complexity $O(\log n)$. We do not go into any details here but refer the reader to [AhHU83] or [CoLR90]. Using these results, we put the vertex set of our digraph into a priority queue T in Dijkstra's algorithm, with d as the priority function. This leads to the following modified algorithm.

Algorithm 3.7.6. Let (G, w) be a given network, where G is a graph or a digraph and all lengths $w(e)$ are nonnegative. We denote the adjacency list of v by A_v. Moreover, let T be a priority queue with priority function d. The algorithm calculates the distances with respect to a vertex s.

Procedure DIJKSTRAPQ$(G, w, s; d)$.

```
(1) T ← {s}, d(s) ← 0;
(2) for s ∈ V \ {s} do d(v) ← ∞ od
(3) while T ≠ ∅ do
(4)       u := min T;
(5)       DELETEMIN (T);
(6)       for v ∈ A_u do
(7)             if d(v) = ∞
(8)             then d(v) ← d(u) + w_uv;
(9)                   insert v with priority d(v) into T
(10)            else if d(u) + w_uv < d(v)
(11)                  then change the priority of v to d(v) ← d(u) + w_uv
(12)                  fi
(13)            fi
(14)      od
(15) od
```

As noted before, each of the operations during the **while** loop can be performed in $O(\log |V|)$ steps, and altogether we need at most $O(|E|) + O(|V|)$ such operations. If G is connected, this gives the following result.

Theorem 3.7.7. *Let (G, w) be a connected network, where w is nonnegative. Then Algorithm 3.7.6 (the modified algorithm of Dijkstra) has complexity $O(|E| \log |V|)$.* □

The discussion above provides a nice example for the fact that sometimes we can decrease the complexity of a graph theoretical algorithm by selecting more appropriate (which usually means more complex) data structures. But this is not a one-way road: conversely, graph theory is a most important tool for implementing data structures. For example, priority queues are usually implemented using a special types of trees (for instance, so-called 2-3-trees). A nice treatment of the close interplay between algorithms from graph theory and data structures may be found in [Tar83].

Exercise 3.7.8. Let s be a vertex of a planar network with a nonnegative length function. What complexity does the calculation of the distances with respect to s have?

Using even more involved data structures, we can further improve the results of Theorem 3.7.7 and Exercise 3.7.8. Implementing a priority queue appropriately (for instance, as a *Fibonacci Heap*), inserting an element or reducing the priority of a given element can be done in $O(1)$ steps; DELETEMIN still requires $O(\log n)$ steps. Thus one may reduce the complexity of Algorithm 3.7.6 to $O(|E| + |V| \log |V|)$; see [FrTa87]. The best theoretical bound known at present is $O(|E| + (|V| \log |V|)/(\log \log |V|))$; see [FrWi94]. This algorithm, however, is of no practical interest as the constants hidden in the big-O notation are too large. If all lengths are relatively small (say, bounded by a constant C), one may achieve a complexity of $O(|E| + |V|(\log C)^{1/2})$; see [AhMOT90]. For the planar case, there is an algorithm with complexity $O(|V|(\log |V|)^{1/2})$; see [Fre87]. A short but nice discussion of various algorithmic approaches of practical interest is in [Ber93]. More information about practical aspects may be found in [GaPa88] and [HuDi88].

To end this section, we present an algorithm which can also treat instances where negative lengths occur, as long as no cycles of negative length exist. This is due to Ford [For56] and Bellman [Bel58].

Exercise 3.7.9. Let (G, w) be a network without cycles of negative length. Show that the following algorithm calculates the distances with respect to a given vertex s and determine its complexity:

Procedure BELLFORD$(G, w, s; d)$

(1) $d(s) \leftarrow 0$;
(2) **for** $v \in V \setminus \{s\}$ **do** $d(v) \leftarrow \infty$ **od**
(3) **repeat**

(4) **for** $v \in V$ **do** $d'(v) \leftarrow d(v)$ **od**
(5) **for** $v \in V$ **do** $d(v) \leftarrow \min(d'(v), \min\{d'(u) + w_{uv} : uv \in E\})$ **od**
(6) **until** $d(v) = d'(v)$ for all $v \in V$.

Apply this algorithm to Example 3.7.4, treating the vertices in the order $1, \ldots, 8$.

3.8 An application: Train schedules

In this section, we discuss a practical problem which can be solved using the algorithm of Dijkstra, namely finding optimal connections in a public transportation system.[4] Such a system consists of several lines (of trains or buses) which are served at regular intervals. Typical examples are the German Intercity network or the American Greyhound bus lines. If someone wants to use such a system to get from one point to another in the network, it may be necessary to change lines a couple of times, each time having to wait for the connection. Often there might be a choice between several routes; we are interested in finding the fastest one. This task is done in practice by interactive information systems, giving travellers the optimal routes to their destinations. For example, the state railway company of the Netherlands uses such a schedule information system based on the algorithm of Dijkstra, as described in [SiTu89]. We now use a somewhat simplified example to illustrate how such a problem can be modelled so that the algorithm of Dijkstra applies. For the sake of simplicity, we we restrict our interpretation to train lines and train stations, and we have our trains begin their runs at fixed intervals. Of course, any set of events occurring *at regular intervals* can be treated similarly.

We begin by constructing a digraph $G = (V, E)$ which has the train stations as vertices and the tracks between two stations as edges. With each edge e, we associate a travel time $f(e)$; here parallel edges might be used to model trains going at different speeds. Edges always connect two consecutive points of a line where the train stops, that is, stations a train just passes through do not occur on this line. Thus lines are just paths or cycles[5] in G. With each line L, we associate a time interval T_L representing the amount of time between two consecutive trains of this line. For each station v on a line L, we define the *time cycle* $t_L(v)$ which specifies at which times the trains of line L leave station v; this is stated modulo T_L. Now let

$$L: \; v_0 \overset{e_1}{\text{——}} v_1 \text{——} \ldots \text{——} v_{n-1} \overset{e_n}{\text{——}} v_n$$

be a line. Clearly, the time of departure at station v_i is the sum of the time of departure at station v_{i-1} and the travelling time $f(e_i)$ from v_{i-1} to v_i, taken

[4] I owe the material of this section to my former student, Dr. Michael Guckert.
[5] Remember the Circle line in the London Underground system!

modulo T_L.[6] Hence the values $t_L(v_i)$ are determined as follows:[7]

$$\begin{aligned} &t_L(v_0) := s_L \pmod{T_L};\\ &t_L(v_i) := t_L(v_{i-1}) + f(e_i) \pmod{T_L} \quad \text{for } i = 1, \ldots, n. \end{aligned} \tag{3.6}$$

The schedule of line L is completely determined by (3.6): the trains depart from station v_i at the time $t_L(v_i)$ (modulo T_L) in intervals of length T_L.

Next we have to calculate the waiting times involved in changing trains. Let $e = uv$ and $e' = vw$ be edges of lines L and L', respectively. A train of line L' leaves the station v at the times

$$t_{L'}(v),\ t_{L'}(v) + T_{L'},\ t_{L'}(v) + 2T_{L'},\ \ldots$$

and a train of line L reaches station v at the times[8]

$$t_L(v),\ t_L(v) + T_L,\ t_L(v) + 2T_L,\ \ldots.$$

Now assume that L and L' have different time cycles. Then the waiting time depends not only on the time cycles, but also on the precise point of time modulo the least common multiple T of T_L and $T_{L'}$. Let us illustrate this by an example. Suppose the time cycle of line L is 12 minutes, while that of L' is 10 minutes so that $T = 60$. For $t_L(v) = 0$ and $t_{L'}(v) = 5$ we get the following schedules at v:

Line L:	0	12	24		36	48
Line L' :	5	15	25	35	45	55

Thus the waiting time for the next train of line L' varies between one minute and nine minutes in this example. To simplify matters, we now assume that all time cycles are actually the same. Then the waiting time at station v is given by

$$w(v_{LL'}) = t_{L'}(v) - t_L(v) \pmod{T}.$$

This even applies in case $L = L'$: then we do not have to change trains.

Exercise 3.8.1. Reduce the case of different time cycles to the special case where all time cycles are equal.

---

[6] We will neglect the amount of time a train stops at station v_i. This can be taken into account by either adding it to the travelling time $f(e_i)$ or by introducing an additional term $w_L(v_i)$ which then has to be added to $t_L(v_{i-1}) + f(e_i)$.

[7] Note that we cannot just put $t_L(v_0) = 0$, as different lines may leave their start stations at different times.

[8] More precisely, the trains of line L leave station v at these times, that is, they reach v a little bit earlier. We assume that this short time interval suffices for the process of changing trains, so that we can leave this out of our considerations as well.

We now construct a further digraph $G' = (V', E')$ which will allow us to find an optimal connection between two stations directly by finding a shortest path. Here a *connection* between two vertices v_0 and v_n in G means a path

$$P:\ v_0 \ \overset{e_1}{\text{———}}\ v_1 \ \text{———}\ \ldots\ \overset{e_n}{\text{———}}\ v_n$$

in G together with the specification of the line L_i corresponding to edge e_i for $i = 1, \ldots, n$, and the travelling time for this connection is

$$f(e_1) + w(v_{L_1L_2}) + f(e_2) + w(v_{L_2L_3}) + \ldots + w(v_{L_{n-1}L_n}) + f(e_n). \qquad (3.7)$$

This suggests the following definition of G'. For each vertex $v \in V$ and each line L serving station v, we have two vertices $(v, L)_{\text{in}}$ and $(v, L)_{\text{out}}$ in V'; for each edge $e = vw$ contained in some line L, there is an edge $(v, L)_{\text{out}}(w, L)_{\text{in}}$ in E'. Moreover, for each vertex v contained in both lines L and L', there is an edge $(v, L)_{\text{in}}(v, L')_{\text{out}}$. Then a directed path from v_0 to v_n in G' corresponds in fact to a connection between v_0 and v_n, and this even includes the information which lines to use and where to change trains. In order to obtain the travelling time (3.7) as the length of the corresponding path in G', we simply define a weight function w' on G' as follows:

$$w'((v, L)_{\text{out}}(w, L)_{\text{in}}) := f(vw)$$
$$w'((v, L)_{\text{in}}(v, L')_{\text{out}}) := w(v_{LL'}).$$

Now our original problem is solved by applying Dijkstra's algorithm to the network (G', w'). Indeed, we may find all optimal connections leaving station v by applying this algorithm (modified as in Exercise 3.7.3) starting from all vertices in (G', w') which have the form $(v, L)_{\text{out}}$.

In this context, let us mention some other problems concerning the *design* of a schedule for several lines having fixed time cycles, that is, the problem of how to choose the times of departure s_L for the lines L for given time cycles T_L. In general, we might want the desired schedule to be optimal with respect to one of the following objectives.

- The longest waiting time (or the sum of all the waiting times) should be minimal.
- The shortest time interval between the departure of two trains from a station should be maximal; that is, we want a safety interval between successive trains.
- The sum of all travelling times between any two stations should be minimal; we might also give each of the routes a weight in this sum corresponding to its importance, maybe according to the expected number of travellers.

These problems are considerably more difficult; in fact, they are NP-hard in general, although polynomial solutions are known when the number of lines is small. We refer to the literature; in particular, for the first two problems

see [Gul80], [Bur86], and [BrBH90]. The last problem was studied in detail by Guckert [Guc96], and the related problem of minimizing the sum of the waiting times of all travellers was treated by Domschke [Dom89]. Both these authors described and tested various heuristics.

3.9 The algorithm of Floyd and Warshall

Sometimes it is not enough to calculate the distances with respect to a certain vertex s in a given network: we need to know the distances between all pairs of vertices. Of course, we may repeatedly apply one of the algorithms treated before, varying the start vertex s over all vertices in V. This results in the following complexities, depending on the specific algorithm used.

algorithm of Moore: $O(|V||E|)$;
algorithm of Dijkstra: $O(|V|^3)$ or $O(|V||E|\log|V|)$;
algorithm of Bellman and Ford: $O(|V|^2|E|)$.

These complexities could even be improved a bit according to the remarks at the end of Section 3.7. Takaoka [Tak92] presented an algorithm with complexity $O(|V|^3(\log\log|V|/\log|V|)^{1/2})$. In the planar case one can achieve a complexity of $O(|V|^2)$; see [Fre87].

In this section, we study an algorithm for this problem which has just the same complexity as the original version of Dijkstra's algorithm, namely $O(|V|^3)$. However, it offers the advantage of allowing some lengths to be negative – though, of course, we cannot allow cycles of negative length. This algorithm is due to Floyd [Flo62], see also Warshall [War62], and works by determining the *distance matrix* $D = (d(v,w))_{v,w\in V}$ of our network.

Algorithm 3.9.1 (Algorithm of Floyd and Warshall). Let (G,w) be a network not containing any cycles of negative length, and assume $V = \{1,\dots,n\}$. Put $w_{ij} = \infty$ if ij is not an edge in G.

Procedure FLOYD$(G,w;d)$

```
(1) for i = 1 to n do
(2)     for j = 1 to n do
(3)         if i ≠ j then d(i,j) ← w_ij else d(i,j) ← 0 fi
(4)     od
(5) od
(6) for k = 1 to n do
(7)     for i = 1 to n do
(8)         for j = 1 to n do
(9)             d(i,j) ← min (d(i,j), d(i,k) + d(k,j))
(10)        od
(11)    od
(12) od
```

Theorem 3.9.2. *Algorithm 3.9.1 computes the distance matrix D for (G, w) with complexity $O(|V|^3)$.*

Proof. The complexity of the algorithm is obvious. Let $D_0 = (d_{ij}^0)$ denote the matrix defined in step (3) and $D_k = (d_{ij}^k)$ the matrix generated during the k-th iteration in step (9). Then D_0 contains all lengths of paths consisting of one edge only. Using induction, it is easy to see that (d_{ij}^k) is the shortest length of a directed path from i to j containing only intermediate vertices from $\{1, \ldots, k\}$. As we assumed that (G, w) does not contain any cycles of negative length, the assertion follows for $k = n$. □

Exercise 3.9.3. Modify algorithm 3.9.1 so that it not only calculates the distance matrix, but also determines shortest paths between any two vertices.

Example 3.9.4. For the network shown in Figure 3.9, the algorithm of Floyd and Warshall computes the accompanying matrices.

Exercise 3.9.5. Apply Algorithm 3.9.1 to the network in Figure 3.8 [Law76].

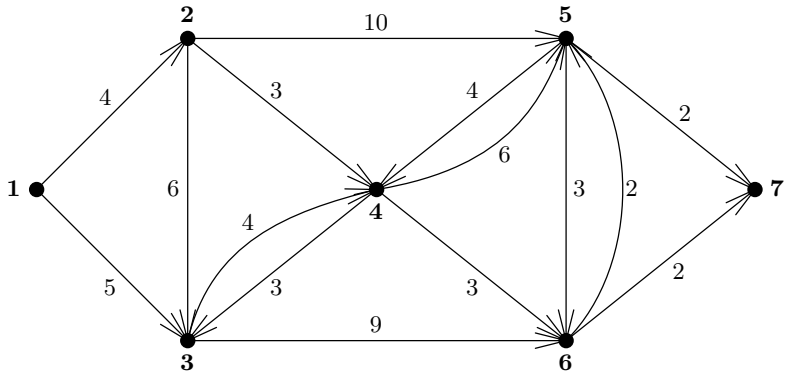


Fig. 3.8. A network

In Section 2.6, we looked at acyclic digraphs associated with partially ordered sets. Such a digraph G is *transitive*: if there is a directed path from u to v, then G has to contain the edge uv. Now let G be an arbitrary acyclic digraph. Let us add the edge uv to G for each pair of vertices (u, v) such that v is accessible from u, but uv is not already an edge . This operation yields the *transitive closure* of G. Clearly, the transitive closure of an acyclic digraph is again acyclic and thus corresponds to a partially ordered set. By definition, two vertices u and v have distance $d(u, v) \neq \infty$ if and only if uv is an edge of the transitive closure of G. Hence the algorithm of Floyd and Warshall can be used to compute transitive closures with complexity $O(|V|^3)$.

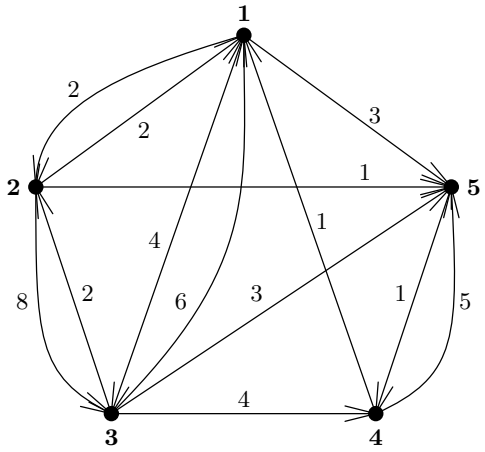


Fig. 3.9. A network

$$D_0 = \begin{pmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & 8 & \infty & 1 \\ 6 & 2 & 0 & 4 & 3 \\ 1 & \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad D_1 = \begin{pmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & 6 & \infty & 1 \\ 6 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & 6 & \infty & 1 \\ 4 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad D_3 = \begin{pmatrix} 0 & 2 & 4 & 8 & 3 \\ 2 & 0 & 6 & 10 & 1 \\ 4 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 2 & 4 & 8 & 3 \\ 2 & 0 & 6 & 10 & 1 \\ 4 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ 2 & 4 & 6 & 1 & 0 \end{pmatrix} \quad D_5 = \begin{pmatrix} 0 & 2 & 4 & 4 & 3 \\ 2 & 0 & 6 & 2 & 1 \\ 4 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ 2 & 4 & 6 & 1 & 0 \end{pmatrix}$$

Exercise 3.9.6. Simplify Algorithm 3.9.1 for computing the transitive closure by interpreting the adjacency matrix of an acyclic digraph as a Boolean matrix; see [War62].

Let us mention a further way of associating an acyclic digraph to a partially ordered set. More generally, consider any acyclic digraph G. If uv is an edge in G and if there exists a directed path of length ≥ 2 from u to v in G, we remove the edge uv from G. This operation yields a digraph called the

transitive reduction G_{red} of G. If G is the digraph associated with a partially ordered set as in Section 2.6, G_{red} is also called the *Hasse diagram* of G. If we want to draw a Hasse diagram, we usually put the vertices of equal rank on the same horizontal level. Figure 3.10 shows the Hasse diagram of the partially ordered set of the divisors of 36. The orientation of the edges is not shown explicitly: it is understood that all edges are oriented from bottom to top. As an exercise, the reader might draw some more Hasse diagrams.

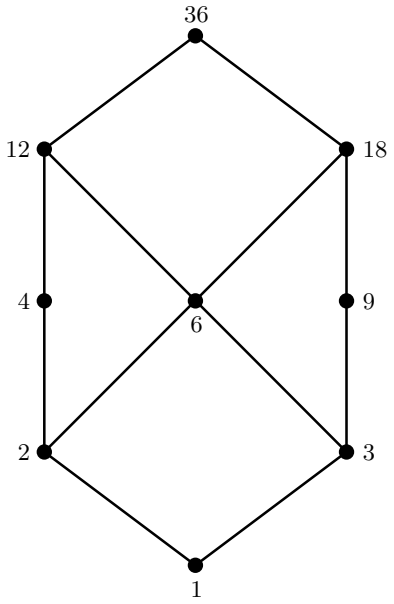


Fig. 3.10. Hasse diagram of the divisors of 36

Exercise 3.9.7. Design an algorithm for constructing the reduction of an acyclic digraph with complexity $O(|V|^3)$ and show that G and G_{red} have the same transitive closure. Hint: Modify the Floyd and Warshall algorithm so that it may be used here to determine longest paths.

For more about the transitive closure and the transitive reduction of an acyclic digraph see [Meh84]. Schnorr [Schn78] gave an algorithm for constructing the transitive closure with an average complexity of $O(|E|)$.

Let us consider a final application of the algorithm of Floyd and Warshall. Sometimes we are interested in finding the *center* of some network.[9] Let (G, w) be a network not containing any cycles of negative length. Then the *eccentricity* of a vertex v is defined as

[9] It is obvious how this notion could be applied in the context of traffic or communication networks.

$$\varepsilon(v) \;=\; \max\{d(v,u)\colon u \in V\}\,.$$

A *center* of a network is a vertex having minimal eccentricity. The centers of a given network can be determined easily using the algorithm of Floyd and Warshall as follows. At the end of the algorithm, $\varepsilon(i)$ simply is the maximum of the i-th row of the matrix $D = (d(i,j))$, and the centers are those vertices for which this maximum is minimal. For example, the vertices of the network of Example 3.9.4 have eccentricities $\varepsilon(1) = 4$, $\varepsilon(2) = 6$, $\varepsilon(3) = 4$, $\varepsilon(4) = 5$ and $\varepsilon(5) = 6$, so that 1 and 3 are centers of the network. It is obvious that the complexity of the additional operations needed – namely finding the required maxima and minima – is dominated by the complexity $O(|V|^3)$ of the algorithm of Floyd and Warshall. Thus we have the following result.

Theorem 3.9.8. *Let N be a network without cycles of negative length. Then the centers of N can be determined with complexity $O(|V|^3)$.* □

If we take all edges in a given graph (directed or not) to have length 1, the above definition yields the eccentricities of the vertices and the centers of the graph in the graph theory sense. Sometimes we are interested in the maximal eccentricity of all vertices of a graph. This value is called the *diameter* of the graph; again, this is a notion of interest in communications networks, see [Chu86]. For more on communication networks, we also refer to [Bie89] and [Ber92]. It is a difficult (in fact, NP-hard) problem to choose and assign centers for networks under the restrictions occurring in practical applications, see [BaKP93].

To close this section, we briefly discuss the *dynamic* variant of the problem of determining shortest paths between any two vertices in a network. Suppose we have found a solution for some optimization problem, using an appropriate algorithm. For some reason, we need to change the input data slightly and find an optimal solution for the modified problem. Can we do so using the optimal solution we know already, without having to run the whole algorithm again? For our problem of finding shortest paths, this means keeping up to date the distance matrix D as well as information needed for constructing shortest paths (as, for example, the matrix $P = (p(i,j))$ used in the solution of Exercise 3.9.3) while inserting some edges or reducing lengths of edges. Compare this procedure with calculating all the entries of the matrices D and P again. If all lengths $w(e)$ are integers in the interval $[1, C]$, it is obvious that at most $O(Cn^2)$ such operations can be performed because an edge may be inserted at most once, and the length of each edge can be reduced at most C times. While a repeated application of the algorithm of Floyd and Warshall for a sequence of such operations would need $O(Cn^5)$ steps, it is also possible to solve the problem with complexity just $O(Cn^3 \log nC)$, using an adequate data structure. If we are treating an instance with graph theoretic distances, that is, for $C = 1$, a sequence of $O(n^2)$ insertions of edges needs only $O(n^3 \log n)$ steps. We refer the reader to [AuIMN91] for this topic.

3.10 Cycles of negative length

Later in this book (when treating flows and circulations in Chapter 10), we will need a method to decide whether a given network contains a directed cycle of negative length; moreover, we should also be able to find such a cycle explicitly. We shall now modify the algorithm of Floyd and Warshall to meet these requirements. The essential observation is as follows: a network (G, w) contains a directed cycle of negative length passing through the vertex i if and only if Algorithm 3.9.1 yields a negative value for $d(i, i)$.

Algorithm 3.10.1. Let (G, w) be a network with vertex set $V = \{1, \ldots, n\}$.
Procedure NEGACYCLE$(G, w; d, p, \text{neg}, K)$

```
(1) neg ← false, k ← 0;
(2) for i = 1 to n do
(3)       for j = 1 to n do
(4)            if i ≠ j then d(i,j) ← w_ij else d(i,j) ← 0 fi
(5)            if i = j or d(i,j) = ∞ then p(i,j) ← 0 else p(i,j) ← i fi
(6)       od
(7) od
(8) while neg = false and k < n do
(9)       k ← k + 1;
(10)      for i = 1 to n do
(11)           if d(i,k) + d(k,i) < 0
(12)           then neg ← true; CYCLE(G, p, k, i; K)
(13)           else for j = 1 to n do
(14)      if d(i,k) + d(k,j) < d(i,j)
(15)                then d(i,j) ← d(i,k) + d(k,j); p(i,j) ← p(k,j)
(16)                fi
(17)           od
(18)           fi
(19)      od
(20) od
```

Here CYCLE denotes a procedure which uses p for constructing a cycle of negative length containing i and k. Note that $p(i, j)$ is, at any given point of the algorithm, the predecessor of j on a – at that point of time – shortest path from i to j. CYCLE can be described informally as follows. First, set $v_0 = i$, then $v_1 = p(k, i)$, $v_2 = p(k, v_1)$, etc., until $v_a = k = p(k, v_{a-1})$ for some index a. Then continue with $v_{a+1} = p(i, k)$, $v_{a+2} = p(i, v_{a+1})$, etc., until an index b is reached for which $v_{a+b} = v_0 = i = p(i, v_{a+b-1})$. Now the cycle we have found uses each edge in the direction opposite to its orientation, so that $(v_{a+b} = v_0, v_{a+b-1}, \ldots, v_1, v_0)$ is the desired directed cycle of negative length through i and k. It can then be stored in a list K. We leave it to the reader to state this procedure in a formally correct way.

If (G, w) does not contain any directed cycles of negative length, the variable neg has value false at the end of Algorithm 3.10.1. In this case, d contains the distances in (G, w) as in the original algorithm of Floyd and Warshall. The matrix $(p(i, j))$ may then be used to find a shortest path between any two given vertices; this is similar to the procedure CYCLE discussed above. Altogether, we get the following result.

Theorem 3.10.2. *Algorithm 3.10.1 decides with complexity $O(|V|^3)$ whether or not a given network (G, w) contains cycles of negative length; in case it does, such a cycle is constructed. Otherwise, the algorithm yields the distance matrix $(d(i, j))$ for (G, w).* □

Exercise 3.10.3. Let G be a digraph on n vertices having a root s, and let w be a length function on G. Modify the algorithm of Bellman and Ford (see Exercise 3.7.9) so that it determines whether (G, w) contains a cycle of negative length. If there is no such cycle, the algorithm should determine an SP-tree with root s using a procedure SPTREE. Write down such a procedure explicitly.

Exercise 3.10.4. Modify the algorithm of Floyd and Warshall so that it determines the shortest length of a directed cycle in a network not containing any cycles of negative length.

3.11 Path algebras

Let (G, w) be a network without cycles of negative length. According to Bellman's equations (Proposition 3.5.1), the distances u_i with respect to a vertex i then satisfy the conditions

$$\text{(B)} \quad u_1 = 0 \quad \text{and} \quad u_i = \min\{u_k + w_{ki} : i \neq k\} \quad \text{for } i = 2, \ldots, n.$$

In this section, we consider the question whether such a system of equations might be solved using methods from linear algebra. In fact, this is possible by introducing appropriate algebraic structures called *path algebras*. We only sketch the basic ideas here; for details we refer to the literature, in particular [Car71, Car79, GoMi84, Zim81].[10]

We begin with a suitable transformation of the system (B). Recall that we put $w_{ij} = \infty$ if ij is not an edge of our network; therefore we extend $\mathbb{R}$ to $\overline{\mathbb{R}} = \mathbb{R} \cup \{\infty\}$. Moreover, we introduce two operations $\oplus$ and $*$ on $\overline{\mathbb{R}}$:

$$a \oplus b := \min(a, b) \qquad \text{and} \qquad a * b := a + b,$$

where, as usual, we define $a + \infty$ to be ∞. Obviously, (B) can be written as

[10] This section is included just to provide some more theoretical background. As it will not be used in the rest of the book, it may be skipped.

$$u_1 = \min\left(0, \min\left\{u_k + w_{k1} : k \neq 1\right\}\right) \qquad \text{and}$$
$$u_i = \min\left(\infty, \min\left\{u_k + w_{ki} : k \neq i\right\}\right),$$

since (G, w) does not contain any cycles of negative length. Using the operations introduced above, we get the system of equations

$$(\mathrm{B}') \qquad u_1 = \bigoplus_{k=1}^{n}(u_k * w_{k1}) \oplus 0, \quad u_i = \bigoplus_{k=1}^{n}(u_k * w_{ki}) \oplus \infty,$$

where we set $w_{ii} = \infty$ for $i = 1, \ldots, n$. We can now define matrices over $\overline{\mathbb{R}}$ and apply the operations $\oplus$ and $*$ to them in analogy to the usual definitions from linear algebra. Then (B′) (and hence (B)) can be written as a linear system of equations:

$$(\mathrm{B}'') \qquad u = u * W \oplus b,$$

where $u = (u_1, \ldots, u_n)$, $b = (0, \infty, \ldots, \infty)$ and $W = (w_{ij})_{i,j=1,\ldots,n}$.

Thus Bellman's equations may be viewed as a linear system of equations over the algebraic structure $(\overline{\mathbb{R}}, \oplus, *)$. Then the algorithm of Bellman and Ford given in Exercise 3.7.9 admits the following interpretation. First set

$$u^{(0)} = b \qquad \text{and then recursively} \qquad u^{(k)} = u^{(k-1)} * W \oplus b,$$

until the sequence eventually converges to $u^{(k)} = u^{(k-1)}$, which in our case occurs for $k = n$ or earlier. Hence the algorithm of Bellman and Ford is analogous to the Jacobi method from classical linear algebra over $\mathbb{R}$; see, for instance, [Str88].

These observations lead to studying algebraic structures which satisfy the same conditions as $(\overline{\mathbb{R}}, \oplus, *)$. A *path algebra* or *dioid* is a triple $(R, \oplus, *)$ such that $(R, \oplus)$ is a commutative monoid, $(R, *)$ is a monoid, and both distributive laws hold; moreover, the neutral element o of $(R, \oplus)$ satisfies the absorption law. This means that the following axioms hold, where e denotes the neutral element for $(R, *)$:

(1) $a \oplus b = b \oplus a$;
(2) $a \oplus (b \oplus c) = (a \oplus b) \oplus c$;
(3) $a \oplus o = a$;
(4) $a * o = o * a = o$;
(5) $a * e = e * a = a$;
(6) $a * (b * c) = (a * b) * c$;
(7) $a * (b \oplus c) = (a * b) \oplus (a * c)$;
(8) $(b \oplus c) * a = (b * a) \oplus (c * a)$.

Exercise 3.11.1. Show that $(\overline{\mathbb{R}}, \oplus, *)$ is a path algebra with $e = 0$ and $o = \infty$.

Exercise 3.11.2. Let $(R, \oplus, *)$ be a path algebra. We define a relation $\succeq$ on R by

$$a \succeq b \quad \Longleftrightarrow \quad a = b \oplus c \quad \text{for some } c \in R.$$

Show that $\succeq$ is a preordering (that is, it is reflexive and transitive). If $\oplus$ is idempotent (that is, $a \oplus a = a$ for all $a \in R$), then $\succeq$ is even a partial ordering (that is, it is also antisymmetric).

Exercise 3.11.3. Let (G, w) be a network without cycles of negative length. Give a matrix equation for the distance matrix $D = (d(i, j))$.

We now transfer the notions developed in the special case of $(\overline{\mathbb{R}}, \oplus, *)$ to arbitrary path algebras. For the remainder of this section, a *network* means a pair (G, w) such that G is a digraph, $w : E \to R$ is a length function, and $(R, \oplus, *)$ is a path algebra. The *length* of a path $P = (v_0, v_1, \ldots, v_n)$ is defined as

$$w(P) := w(v_0v_1) * w(v_1v_2) * \ldots * w(v_{n-1}v_n).$$

The *AP-problem* (short for *algebraic path problem*) requires calculating the sums

$$w^*_{ij} = \oplus\, w(P) \qquad \text{(where } P \text{ is a directed path from } i \text{ to } j\text{)}$$

and finding a path P from i to j such that $w(P) = w^*_{ij}$ (if the above sum and such a path exist). For the case $(\overline{\mathbb{R}}, \oplus, *)$, the AP-problem reduces to the familiar *SP-problem* (*shortest path problem*) of determining the distances and shortest paths.

As before, we introduce a matrix $W = (w_{ij})$ whose (i, j)-entry is the length $w(ij)$ if ij is an edge of G. We set $w_{ii} = o$ for $i = 1, \ldots, n$ and $w_{ij} = o$ if $i \neq j$ and ij is not an edge in G. Note that, for the special case $(\overline{\mathbb{R}}, \oplus, *)$ above, we looked at the matrix $W' = W \oplus E$; see Exercise 3.11.5 below. Here E denotes the *unit matrix*, that is, $e_{ii} = e$ for $i = 1, \ldots, n$ and $e_{ij} = o$ for $i \neq j$. As usual, we write A^k for the k-th power of A; moreover, we define $A^{(k)} := E \oplus A \oplus A^2 \oplus \ldots \oplus A^k$.

Lemma 3.11.4. *The (i, j)-entry of the matrix W^k or of $W^{(k)}$ is the sum $\oplus\, w(P)$ over all directed walks from i to j consisting of exactly k edges for the former, and of at most k edges for the latter.*

Proof. Use induction on k. □

We look again at the special case of the SP-problem. In a network (G, w) not containing cycles of negative length, distances can always be realized by paths, so that we need at most $n - 1$ edges. Thus we have $D = W^{(n-1)}$; moreover, $W^{(n-1)} = W^{(n)} = \ldots$. It is easy to see that $W^{(n-1)}$ indeed satisfies the matrix equation given in the solution to Exercise 3.11.3:

$$\begin{aligned} W^{(n-1)} * W \oplus E &= (E \oplus W \oplus \ldots \oplus W^{n-1}) * W \oplus E \\ &= E \oplus W \oplus \ldots \oplus W^{n-1} \oplus W^n = W^{(n)} = W^{(n-1)}. \end{aligned}$$

An element a with $a^{(p)} = a^{(p+1)}$ for some p is called a *stable* element; this notion is important also for general path algebras. In fact, the matrix $W^* = (w^*_{ij})$ of the AP-problem is an infinite sum $\oplus W^k = E \oplus W \oplus W^2 \oplus \ldots$, that is, it is the *limit* of the matrices $W^{(k)}$ for $k \to \infty$. If W is stable, these formulas make sense: if $W^{(p)} = W^{(p+1)}$, then $W^* = W^{(p)}$. That is the reason why criteria for stability play an important part in the theory of path algebras; see [Zim81]. For the theory of convergence, see also [KuSa86].

Exercise 3.11.5. Let $(R, \oplus, *)$ be a path algebra such that $\oplus$ is idempotent. For every matrix A, we put $A' := E \oplus A$. Show $(A')^k = A^{(k)}$ and use this fact to find a technique for calculating $A^{(n)}$; also discuss its complexity.

Now suppose that W is stable; we call $W^* = W^{(p)} = W^{(p+1)}$ the *quasi-inverse* of W. As in the special case $R = \overline{\mathbb{R}}$ above, we have

$$W^* = W^* * W \oplus E = W * W^* \oplus E.$$

Thus, for an arbitrary matrix B , the matrices $Y := W^* * B$ and $Z := B * W^*$, respectively, are solutions of the equations

$$Y = W * Y \oplus B \qquad \text{and} \qquad Z = Z * W \oplus B. \tag{3.8}$$

In particular, we can choose a column or row vector b for B and obtain a linear system of equations analogous to the system (B″).

Exercise 3.11.6. Let $(R, \oplus, *)$ be an arbitrary path algebra. Show that the $(n \times n)$-matrices over R also form a path algebra and define a preordering (or, in the idempotent case, a partial ordering) on this path algebra; see Exercise 3.11.2. Prove that $W^* * B$ and $B * W^*$ are minimal solutions of equation (3.8) and that the system (3.8) has a unique minimal solution in the idempotent case.

Equations having the same form as (3.8) can be solved using techniques analogous to the well-known methods of linear algebra over $\mathbb{R}$. We have already seen that the algorithm of Bellman and Ford corresponds to the Jacobi method; this technique can also be used for the general case of a stable matrix W over any path algebra R. Similarly, it can be shown that the algorithm of Floyd and Warshall corresponds to the Gauss-Jordan elimination method. For more on this result and other general algorithms for solving (3.8), we refer to [GoMi84] and [Zim81].

We conclude this section with some examples which will show that the abstract concept of path algebras makes it possible to treat various interesting network problems with just one general method. However, the SP-problem is still the most important example; here the case of positive lengths – that is, the path algebra $(\overline{\mathbb{R}}_+, \min, +)$ – was already studied by [Shi75]. Similarly, longest paths can be treated using the path algebra $(\mathbb{R} \cup \{-\infty\}, \max, +)$ instead.

Example 3.11.7. Consider the path algebra ($\{0,1\}$, max, min) – that is, the Boolean algebra on two elements – and put $w_{ij} = 1$ for each edge of G. Then Lemma 3.11.4 has the following interpretation. There exists a directed walk from i to j consisting of exactly k edges if and only if the (i,j)-entry of W^k is 1, and of at most k edges if and only if the (i,j)-entry of $W^{(k)}$ is 1. Moreover, the matrix $W^* = W^{(n-1)}$ is the adjacency matrix of the transitive closure of G; see Exercise 3.9.6.

Example 3.11.8. Consider the path algebra ($\overline{\mathbb{R}}_+$, max, min) and think of the length $w(ij)$ of an edge ij as its *capacity*. Then $w(P)$ is the *capacity* of the path P; that is, $w(P)$ is the minimum of the capacities of the edges contained in P. Here the (i,j)-entry of W^k is the largest capacity of a walk from i to j with exactly k edges, while for $W^{(k)}$ it is the largest capacity of a walk from i to j with at most k edges. Hence $W^* = W^{(n-1)}$ and w^*_{ij} is the largest capacity of a walk from i to j; see [Hu61].

Example 3.11.9. Consider the path algebra $(\mathbb{N}_0, +, \cdot)$, where each edge of G has length $w(ij) = 1$. Then W is just the adjacency matrix of G. The (i,j)-entry of W^k and of $W^{(k)}$ represent the number of walks from i to j consisting respectively of precisely and at most k edges; see Exercise 2.2.5. Note that W^* does not exist in general, as there might be infinitely many walks from i to j. If G is an acyclic digraph, W^* is well-defined; in this case $W^* = W^{(n-1)}$ and w^*_{ij} is the number of all directed walks from i to j.

Exercise 3.11.10. Find a path algebra which is suitable for treating the problem of Example 3.1.2, where $w(i,j)$ is the probability $p(i,j)$ described in Example 3.1.2; see [Kal60].

Exercise 3.11.11. Any commutative field $(K, +, \cdot)$ is obviously also a path algebra. Show that A is stable over K if G is acyclic, and give a formula for A^* under this condition. Does the converse hold?

The reader can find a lot of further examples for path algebras in the literature quoted above, in particular in [GoMi84] and in [Zim81]; see also [KuSa86] for applications in automata theory. Finally, let us also mention a practical example from operations research.

Example 3.11.12. We construct a digraph G whose vertices are the single parts, modules, and finished products occurring in an industrial process. We want the edges to signify how many single parts or intermediary modules are needed for assembling bigger modules or finished products. That is, we assign weight $w(i,j)$ to edge ij if we need $w(i,j)$ units of part i for assembling product j. G is called the *gozinto graph*. In most cases, the modules and products are divided into levels of the same rank, where the finished products have highest rank, and basic parts (which are not assembled from any smaller parts) lowest rank; that is, the products and modules are divided into *disposition levels*. The notion of ranks used here is the same as in Exercise 3.6.3; it can be calculated

as in Exercise 3.6.4. Often the gozinto graph is taken to be reduced in the sense of Section 3.9, that is, it contains an edge ij only if part i is used directly in assembling module j, without any intermediate steps. Note that the reduced graph G_{red} can be determined as in Exercise 3.9.7, as we always assume G to be acyclic.[11]

Now suppose that we have a gozinto graph which is reduced already. Sometimes one wants to know how much of each part is needed, no matter whether directly or indirectly. For this purpose, we consider the path algebra $(\mathbb{N}_0, +, \cdot)$ and the given weights $w(ij)$. As G is acyclic, there are only finitely many directed paths from vertex i to vertex j; thus the matrix W^* $(= W^{(n-1)})$ exists. Now it is easily seen that the entry w^*_{ij} is just the total number of units of i needed for the assembly of j. The matrix W^* may, for example, be determined using the algorithm of Bellman and Ford – that is, the generalized Jacobi method – or the analogue of the algorithm of Floyd and Warshall; see [Mue69].

More about gozinto graphs can be found in the book by Müller-Merbach [Mue73] as well as in his two papers already cited. Note that the entries of a column of W^* give the numbers of parts and modules needed for the corresponding product, whereas the entries in the rows show where (and how much of) the corresponding part or module is needed.

[11] This assumption does not always hold in practice. For instance, gozinto graphs containing cycles are quite common in chemical production processes; see [Mue66].

4

Spanning Trees

I think that I shall never see
A poem lovely as a tree.

Joyce Kilmer

In this chapter, we will study trees in considerably more detail than in the introductory Section 1.2. Beginning with some further characterizations of trees, we then present another way of determining the number of trees on n vertices which actually applies more generally: it allows us to compute the number of spanning trees in any given connected graph. The major part of this chapter is devoted to a network optimization problem, namely to finding a spanning tree for which the sum of all edge lengths is minimal. This problem has many applications; for example, the vertices might represent cities we want to connect to a system supplying electricity; then the edges represent the possible connections and the length of an edge states how much it would cost to build that connection. Other possible interpretations are tasks like establishing traffic connections (for cars, trains or planes: the *connector problem*) or designing a network for TV broadcasts. Finally, we consider Steiner trees (these are trees where it is allowed to add some new vertices) and arborescences (directed trees).

4.1 Trees and forests

We defined a tree to be a connected acyclic graph and gave some equivalent conditions in Theorem 1.2.8. The following lemma provides further characterizations for trees.

Lemma 4.1.1. *Let G be a graph. Then the following four conditions are equivalent:*

(1) *G is a tree.*
(2) *G does not contain any cycles, but adding any further edge yields a cycle.*
(3) *Any two vertices of G are connected by a unique path.*
(4) *G is connected, and any edge of G is a bridge.*

Proof. First let G be a tree. We add any new edge, say $e = uv$. Since G is connected, there is a path W from v to u. Then

$$v \overset{W}{\text{———}} u \overset{e}{\text{———}} v$$

is a cycle. As G itself is acyclic by definition, condition (1) implies (2).

Next assume the validity of (2) and let u and v be any two vertices of G. Suppose there is no path between u and v. Then u and v are not adjacent; also, adding the edge uv to G cannot yield a cycle, contradicting (2). Thus G must be connected. Now suppose that G contains two different paths W and W' from u to v. Obviously, following W from u to v and then W' (in reverse order) from v to u would give a closed walk in G. But then G would have to contain a cycle, a contradiction. Hence condition (2) implies (3).

Now assume the validity of (3) and let $e = uv$ be any edge in G. Suppose e is not a bridge so that $G \setminus e$ is still connected. But then there exist two disjoint paths from u to v in G. This contradiction establishes (4).

Finally, assume the validity of (4). Suppose G contains a cycle C. Then any edge of C could be omitted from G, and the resulting graph would still be connected. In other words, no edge of C would be a bridge. This contradiction proves (1). □

Exercise 4.1.2. A connected graph is called *unicyclic* if it contains exactly one cycle. Show that the following statements are equivalent [AnHa67]:

(1) G is unicyclic.
(2) $G \setminus e$ is a tree for a suitable edge e.
(3) G is connected, and the number of vertices is the same as the number of edges.
(4) G is connected, and the set of all edges of G which are not bridges forms a cycle.

Exercise 4.1.3. Prove that every tree has either exactly one center or exactly two centers; see Section 3.9. Discuss the relationship between the eccentricity and the diameter of a tree.

Exercise 4.1.4. Let G be a forest with exactly $2k$ vertices of odd degree. Prove that the edge set of G is the disjoint union of k paths.

Exercise 4.1.5. Let T be a tree, and suppose that the complementary graph $\overline{T}$ is not connected. Describe the structure of T and show that these graphs $\overline{T}$ are precisely the disconnected graphs with the maximal number of edges.

Exercise 4.1.6. Determine all isomorphism classes of trees on six vertices and calculate the number of trees in each isomorphism class, as well as the number of all trees on six vertices. Moreover, find the corresponding automorphism groups.

We note that the number t_n of isomorphism classes of trees on n vertices grows very rapidly with n, a phenomenon illustrated by Table 4.1 which is taken from [Har69]; for $n = 1, 2, 3$, trivially $t_n = 1$. Harary also develops a remarkable formula for the t_n which is due to Otter [Ott48]; as this uses the method of generating functions, it is beyond the scope of the present book.

Table 4.1. Number t_n of isomorphism classes for trees on n vertices

n	4	5	6	7	8	9	10
t_n	2	3	6	11	23	47	106
n	11	12	13	14	15	16	17
t_n	235	551	1301	3159	7741	19320	48629
n	18	19	20	21	22	23	24
t_n	123867	317955	832065	2144505	5623756	14828074	39299897

We refer the reader to [CaRa91] for an interesting exposition of the problem of checking whether or not two given rooted trees are isomorphic. Here a *rooted tree* is just a tree T with a distinguished vertex r which is called the *root* of T; this terminology makes sense as T has a unique orientation so that r indeed becomes the root of the resulting directed tree.

4.2 Incidence matrices

In this section we consider a further matrix associated with a given digraph. This will be used for yet another characterization of trees and for finding a formula for the number of spanning trees of an arbitrary connected graph.

Definition 4.2.1. Let G be a digraph with vertex set $V = \{1, \ldots, n\}$ and edge set $E = \{e_1, \ldots, e_m\}$. Then the $n \times m$ matrix $M = (m_{ij})$, where

$$m_{ij} = \begin{cases} -1 & \text{if } i \text{ is the tail of } e_j, \\ 1 & \text{if } i \text{ is the head of } e_j, \\ 0 & \text{otherwise,} \end{cases}$$

is called the *incidence matrix* of G.

Of course, M depends on the labelling of the vertices and edges of G; thus it is essentially only determined up to permutations of its rows and columns. For example, the digraph of Figure 2.1 has the following incidence matrix, if we number the vertices and edges as in 2.2.1:

$$\begin{pmatrix} -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 0 & -1 & 0 & 0 \end{pmatrix}$$

Note that each column of an incidence matrix contains exactly two non-zero entries, namely one entry 1 and one entry -1; summing the entries -1 in row i gives $d_{\text{out}}(i)$, whereas summing the entries 1 yields $d_{\text{in}}(i)$. The entries $0, 1$ and -1 are often considered as integers, and the matrix M is considered as a matrix over $\mathbb{Z}$, $\mathbb{Q}$ or $\mathbb{R}$. We could also use any other ring R as long as $1 \neq -1$, that is, R should have characteristic $\neq 2$.

Lemma 4.2.2. *Let G be a digraph with n vertices. Then the incidence matrix of G has rank at most $n - 1$.*

Proof. Adding all the rows of the incidence matrix gives a row for which all entries equal 0. □

We will soon determine the precise rank of the incidence matrix. To this end, we first characterize the forests among the class of all digraphs; of course, a digraph G is called a *forest* if the undirected version $|G|$ is a forest, as in the special case of trees.

Theorem 4.2.3. *A digraph G with incidence matrix M is a forest if and only if the columns of M are linearly independent.*

Proof. We have to show that G contains a cycle if and only if the columns of M are linearly dependent. Suppose first that

$$C \quad = \quad v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \dots \xrightarrow{e_k} v_k$$

is a cycle in G, and let $s_1, \dots, s_k$ be the columns of M corresponding to the edges $e_1, \dots, e_k$. Moreover, let $x_i = 1$ if e_i is a forward edge, and $x_i = -1$ if e_i is a backward edge in C (for $i = 1, \dots, k$). Then $x_1 s_1 + \dots + x_k s_k = 0$.

Conversely, let the columns of M be linearly dependent. Then there are columns $s_1, \dots, s_k$ and integers $x_1, \dots, x_k \neq 0$ such that $x_1 s_1 + \dots + x_k s_k = 0$. Let E' be the set of edges corresponding to the columns $s_1, \dots, s_k$ and V' the set of vertices of G incident with the edges contained in E', and write $G' = (V', E')$. Note that every vertex of the associated graph $|G'|$ has degree at least 2. Now Exercise 1.2.5 shows that no connected component of $|G'|$ is a tree. Hence all components of $|G'|$ contain cycles, so that $|G|$ cannot be a forest. □

Theorem 4.2.4. *Let G be a digraph with n vertices and p connected components. Then the incidence matrix M of G has rank $n - p$.*

Proof. According to Theorem 4.2.3, the rank of M is the number of edges of a maximal forest T contained in $|G|$. If $p = 1$, T is a tree and has exactly $n - 1$ edges; thus M has rank $n - 1 = n - p$ in this case.

Now suppose $p \neq 1$. Then G can be partitioned into its p connected components, that is, T is the disjoint union of p trees. Suppose that these trees have $n_1, \ldots, n_p$ vertices, respectively. Then the incidence matrix of G has rank $(n_1 - 1) + \ldots + (n_p - 1) = n - p$. □

Next we want to show that the incidence matrix of a digraph has a very special structure. We require a definition. A matrix over $\mathbb{Z}$ is called *totally unimodular* if each square submatrix has determinant 0, 1 or -1. These matrices are particularly important in combinatorial optimization; for example, the famous theorem about integral flows in networks[1] is a consequence of the following result; see also [Law76], §4.12.

Theorem 4.2.5. *Let M be the incidence matrix of a digraph G. Then M is totally unimodular.*

Proof. Let M' be any square submatrix of M, say with k rows and columns. We shall use induction on k. Trivially, M' has determinant 0, 1 or -1 if $k = 1$. So let $k \neq 1$. Assume first that each column of M' contains two non-zero entries. Then the rows and columns of M' define a digraph G' with k vertices and k edges. By Theorem 1.2.7, $|G'|$ cannot be acyclic, so that G' is not a forest. Therefore the columns of M' are linearly dependent by Theorem 4.2.3 and hence $\det M' = 0$. Finally assume that there is a column of M' with at most one entry $\neq 0$. We may calculate the determinant of M' by expanding it with respect to such a column. Then we obtain a factor 0, 1, or -1 multiplied with the determinant of a square $((k-1) \times (k-1))$-submatrix M''. The assertion follows by induction. □

Corollary 4.2.6. *Let G be a digraph with n vertices and $n - 1$ edges. Let B be the matrix which arises from the incidence matrix M of G by deleting an arbitrary row. If G is a tree, then* $\det B = 1$ *or* $\det B = -1$*, and otherwise* $\det B = 0$.

Proof. Note that the row deleted from M is a linear combination of the remaining rows. By Theorem 4.2.4, B has rank $n - 1$ if and only if G is a tree. Now the assertion is an immediate consequence of Theorem 4.2.5. □

Next we use the incidence matrix to determine the number of spanning trees of a digraph G. Of course, a *spanning tree* of G is just a directed subgraph T of G such that $|T|$ is a spanning tree for $|G|$.

Theorem 4.2.7 (matrix tree theorem). *Let B be the matrix arising from the incidence matrix of a digraph G by deleting an arbitrary row. Then the number of spanning trees of G is* $\det BB^T$.

[1] We will treat this result in Chapter 6. Actually we shall use a different proof which is not based on Theorem 4.2.5.

Proof. Let n be the number of vertices of G. For any set S of $n-1$ column indices, we denote the matrix consisting of the $n-1$ columns of B corresponding to S by B_S. Now the theorem of Cauchy and Binet (see, for instance, [Had61]) implies

$$\det BB^T \;=\; \sum_S \det B_S B_S^T \;=\; \sum_S (\det B_S)^2.$$

By Corollary 4.2.6, $\det B_S \neq 0$ if and only if the edges of G corresponding to S form a tree; moreover, in this case, $(\det B_S)^2 = 1$. This proves the assertion. □

Theorem 4.2.7 is contained implicitly in [Kirh47]. Not surprisingly, this result may also be used to determine the number of spanning trees of a graph G by considering the incidence matrix of any orientation of G. We need the following simple lemma; then the desired result is an immediate consequence of this lemma and Theorem 4.2.7.

Lemma 4.2.8. *Let A be the adjacency matrix of a graph G and M the incidence matrix of an arbitrary orientation H of G, where both matrices use the same ordering of the vertices for numbering the rows and columns. Then $MM^T = \operatorname{diag}(\deg 1, \ldots, \deg n) - A$.*

Proof. The (i,j)-entry of MM^T is the inner product of the i-th and the j-th row of M. For $i \neq j$, this entry is -1 if ij or ji is an edge of H and 0 otherwise. For $i = j$, we get the degree $\deg i$. □

Theorem 4.2.9. *Let A be the adjacency matrix of a graph G and A' the matrix $-A + \operatorname{diag}(\deg 1, \ldots, \deg n)$. Then the number of spanning trees of G is the common value of all minors of A' which arise by deleting a row and the corresponding column from A'.* □

In Section 4.8, we will give a different proof for Theorem 4.2.9 which avoids using the theorem of Cauchy and Binet. The matrix A' is called the *degree matrix* or the *Laplacian matrix* of G. As an example, let us consider the case of complete graphs and thus give a third proof for Corollary 1.2.11.

Example 4.2.10. Theorem 4.2.9 contains a formula for the number of all trees on n vertices; note that this formula counts the different trees, not the isomorphism classes of trees. Obviously, the degree matrix of K_n is $A' = nI - J$, where J is the matrix having all entries = 1. By Theorem 4.2.9, the number of trees on n vertices is the determinant of a minor of A', that is

$$\begin{vmatrix} n-1 & -1 & \ldots & -1 \\ -1 & n-1 & \ldots & -1 \\ \ldots & \ldots & \ldots & \ldots \\ -1 & -1 & \ldots & n-1 \end{vmatrix} = \begin{vmatrix} n-1 & -n & -n & \ldots & -n \\ -1 & n & 0 & \ldots & 0 \\ -1 & 0 & n & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ -1 & 0 & 0 & \ldots & n \end{vmatrix}$$

$$= \begin{vmatrix} 1 & 0 & 0 & \dots & 0 \\ -1 & n & 0 & \dots & 0 \\ -1 & 0 & n & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ -1 & 0 & 0 & \dots & n \end{vmatrix}$$

$$= n^{n-2}.$$

The following exercise concerns a similar application of the matrix tree theorem; see [FiSe58]. A simple direct proof can be found in [Abu90] where this result is also used to give yet another proof for Corollary 1.2.11.

Exercise 4.2.11. Use Theorem 4.2.9 to show that the number of spanning trees of the complete bipartite graph $K_{m,n}$ is $m^{n-1}n^{m-1}$.

Note that we can also define incidence matrices for graphs: the matrix M has entry $m_{ij} = 1$ if vertex i is incident with edge e_j, and $m_{ij} = 0$ otherwise. But the statements analogous to Lemma 4.2.2 and Theorem 4.2.3 do not hold; for example, the three columns of a cycle of length 3 are linearly independent over $\mathbb{Z}$. However, the situation changes if we consider the incidence matrix M as a matrix over $\mathbb{Z}_2$.

Exercise 4.2.12. Prove the analogues of 4.2.2 through 4.2.4 for graphs, where M is considered as a binary matrix.

The incidence matrix M of a graph – considered as a matrix over the integers – is not unimodular in general, as the following exercise shows. Moreover, it provides a further important characterization of bipartite graphs.

Exercise 4.2.13. Let G be a graph with incidence matrix M. Show that G is bipartite if and only if M is totally unimodular as a matrix over $\mathbb{Z}$.

Hint: The proof that unimodularity of M is necessary is similar to the proof of Theorem 4.2.5. The converse can be proved indirectly.

Exercise 4.2.14. Let e be an edge of K_n. Determine the number of spanning trees of $K_n \setminus e$.

Exercise 4.2.15. Let G be a forest with n vertices and m edges. How many connected components does G have?

Sometimes, a list of all spanning trees of a given graph is needed, or an arbitrary choice of some spanning tree of G (a *random* spanning tree). These tasks are treated in [CoDN89]; in particular, it is shown that the latter problem can be solved with complexity $O(|V|^3)$.

4.3 Minimal spanning trees

In this section, we consider spanning forests in networks. Thus let (G, w) be a network. For any subset T of the edge set of G, we define the *weight* of T by

$$w(T) = \sum_{e \in T} w(e).$$

A spanning forest of G is called a *minimal spanning forest* if its weight is minimal among all the weights of spanning forests; similarly, a *minimal spanning tree* has minimal weight among spanning trees. We restrict ourselves to spanning trees; the general case can be treated by considering a minimal spanning tree for each connected component of G. Thus, we now assume G to be connected.

Minimal spanning trees were first considered by Boruvka [Bor26a, Bor26b]. Shortly after 1920, electricity was to be supplied to the rural area of Southern Moravia; the problem of finding as economical a solution as possible for the proposed network was presented to Boruvka. He found an algorithm for constructing a minimal spanning tree and published it in the two papers cited above. We will present his algorithm in the next section. Boruvka's papers were overlooked for a long time; often the solution of the minimal spanning tree problem is attributed to Kruskal and Prim [Kru56, Pri57], although both of them quote Boruvka; see the interesting article [GrHe85] for a history of this problem. There one also finds references to various applications reaching from the obvious examples of constructing traffic or communication networks to more remote ones in classification problems, automatic speech recognition, image processing, etc.

As the orientation of edges is insignificant when looking at spanning trees, we may assume that G is a graph. If the weight function w should be constant, every spanning tree is minimal; then such a tree can be found with complexity $O(|E|)$ using a BFS, as described in Section 3.3. For the general case, we shall give three efficient algorithms in the next section. Corollary 1.2.11 and Exercise 4.2.11 show that the examination of all spanning trees would be a method having non-polynomial complexity.

But first we characterize the minimal spanning trees. Let us introduce the following notation. Consider a spanning tree T and an edge e not contained in T. By Lemma 4.1.1, the graph arising from T by adding e contains a unique cycle; we denote this cycle by $C_T(e)$. The following result is of fundamental importance.

Theorem 4.3.1. *Let (G, w) be a network, where G is a connected graph. A spanning tree T of G is minimal if and only if the following condition holds for each edge e in $G \setminus T$:*

$$w(e) \geq w(f) \qquad \text{for every edge } f \text{ in } C_T(e). \tag{4.1}$$

Proof. First suppose that T is minimal. If (4.1) is not satisfied, there is an edge e in $G \setminus T$ and an edge f in $C_T(e)$ with $w(e) < w(f)$. Removing f from T splits T into two connected components, since f is a bridge. Adding e to $T \setminus f$ gives a new spanning tree T'; as $w(e) < w(f)$, T' has smaller weight than T. This contradicts the minimality of T.

Conversely, suppose that (4.1) is satisfied. We choose some minimal spanning tree T' and show $w(T) = w(T')$, so that T is minimal as well. We use induction on the number k of edges in $T' \setminus T$. The case $k = 0$ (that is, $T = T'$) is trivial. Thus let e' be an edge in $T' \setminus T$. Again, we remove e' from T', so that T' splits into two connected components V_1 and V_2. If we add the path $C_T(e') \setminus \{e'\}$ to $T' \setminus \{e'\}$, V_1 and V_2 are connected again. Hence $C_T(e')$ has to contain an edge e connecting a vertex in V_1 to a vertex in V_2. Note that e cannot be an edge of T', because otherwise $T' \setminus \{e'\}$ would still be connected. The minimality of T' implies $w(e) \geq w(e')$: replacing e' by e in T', we obtain another spanning tree T''; and if $w(e) < w(e')$, this tree would have smaller weight than T', a contradiction. On the other hand, by condition (4.1), $w(e') \geq w(e)$; hence $w(e') = w(e)$ and $w(T'') = w(T')$. Thus T'' is a minimal spanning tree as well. Note that T'' has one more edge in common with T than T'; using induction, we conclude $w(T) = w(T'') = w(T')$. □

Next we give another characterization of minimal spanning trees. To do so, we need two definitions. Let G be a graph with vertex set V. A *cut* is a partition $S = \{X, X'\}$ of V into two nonempty subsets. We denote the set of all edges incident with one vertex in X and one vertex in X' by $E(S)$ or $E(X, X')$; any such edge set is called a *cocycle*. We will require cocycles constructed from trees:

Lemma 4.3.2. *Let G be a connected graph and T a spanning tree of G. For each edge e of T, there is exactly one cut $S_T(e)$ of G such that e is the only edge which T has in common with the corresponding cocycle $E(S_T(e))$.*

Proof. If we remove e from T, the tree is divided into two connected components and we get a cut $S_T(e)$. Obviously, the corresponding cocycle contains e, but no other edge of T. It is easy to see that this is the unique cut with the desired property. □

Theorem 4.3.3. *Let (G, w) be a network, where G is a connected graph. A spanning tree T of G is minimal if and only if the following condition holds for each edge $e \in T$:*

$$w(e) \leq w(f) \qquad \text{for every edge } f \text{ in } E(S_T(e)). \tag{4.2}$$

Proof. First let T be minimal. Suppose that there is an edge e in T and an edge f in $E(S_T(e))$ with $w(e) > w(f)$. Then, by removing e from T and adding f instead, we could construct a spanning tree of smaller weight than T, a contradiction.

Conversely, suppose that (4.2) is satisfied. We want to reduce the statement to Theorem 4.3.1; thus we have to show that condition (4.1) is satisfied. Let e be an edge in $G \setminus T$ and $f \neq e$ an edge in $C_T(e)$. Consider the cocycle $E(S_T(f))$ defined by f. Obviously, e is contained in $E(S_T(f))$; hence (4.2) yields $w(f) \leq w(e)$. □

Exercise 4.3.4. Let (G, w) be a network, and let v be any vertex. Prove that every minimal spanning tree has to contain an edge incident with v which has smallest weight among all such edges.

Exercise 4.3.5. Let (G, w) be a network, and assume that all edges have distinct weights. Show that (G, w) has a *unique* minimal spanning tree. [Bor26a]

Exercise 4.3.6. Let (G, w) be a network. Show that all minimal spanning trees of (G, w) have the same weight sequence. [Kra07]

Hint: Review the proof of Theorem 4.3.1.

4.4 The algorithms of Prim, Kruskal and Boruvka

In this section, we will treat three popular algorithms for determining minimal spanning trees, all of which are based on the characterizations given in the previous section. Let us first deal with a generic algorithm which has the advantage of allowing a rather simple proof. The three subsequent algorithms are special cases of this general method which is due to Prim [Pri57].

Algorithm 4.4.1. Let $G = (V, E)$ be a connected graph with vertex set $V = \{1, \ldots, n\}$ and $w\colon E \to \mathbb{R}$ a weight function for G. The algorithm constructs a minimal spanning tree T for (G, w).

Procedure MINTREE$(G, w; T)$

(1) **for** $i = 1$ **to** n **do** $V_i \leftarrow \{i\}$; $T_i \leftarrow \emptyset$ **od**
(2) **for** $k = 1$ **to** $n - 1$ **do**
(3) choose V_i with $V_i \neq \emptyset$;
(4) choose an edge $e = uv$ with $u \in V_i$, $v \notin V_i$, and $w(e) \leq w(e')$ for all edges $e' = u'v'$ with $u' \in V_i$, $v' \notin V_i$;
(5) determine the index j for which $v \in V_j$;
(6) $V_i \leftarrow V_i \cup V_j$; $V_j \leftarrow \emptyset$;
(7) $T_i \leftarrow T_i \cup T_j \cup \{e\}$; $T_j \leftarrow \emptyset$;
(8) **if** $k = n - 1$ **then** $T \leftarrow T_i$ **fi**
(9) **od**

Theorem 4.4.2. *Algorithm 4.4.1 determines a minimal spanning tree for the network* (G, w).

Proof. We use induction on $t := |T_1| + \ldots + |T_n|$ to prove the following claim:

$$\text{For } t = 0, \ldots, n-1, \text{ there exists a minimal spanning tree } T \text{ of } G \text{ containing } T_1, \ldots, T_n. \tag{4.3}$$

For $t = n-1$, this claim shows that the algorithm is correct. Clearly, (4.3) holds at the beginning of the algorithm – before the loop (2) to (9) is executed for the first time – since $t = 0$ at that point. Now suppose that (4.3) holds for $t = k-1$, that is, before the loop is executed for the k-th time. Let $e = uv$ with $u \in V_i$ be the edge which is constructed in the k-th iteration. If e is contained in the tree T satisfying (4.3) for $t = k-1$, there is nothing to show. Thus we may assume $e \notin T$. Then $T \cup \{e\}$ contains the unique cycle $C = C_T(e)$; obviously, C has to contain another edge $f = rs$ with $r \in V_i$ and $s \notin V_i$. By Theorem 4.3.1, $w(e) \geq w(f)$. On the other hand, by the choice of e in step (4), $w(e) \leq w(f)$. Hence $w(e) = w(f)$, and $T' = (T \cup \{e\}) \setminus \{f\}$ is a minimal spanning tree of G satisfying (4.3) for $t = k$. □

Of course, we cannot give the precise complexity of Algorithm 4.4.1: this depends both on the choice of the index i in step (3) and on the details of the implementation. We now turn to the three special cases of Algorithm 4.4.1 mentioned above. All of them are derived by making steps (3) and (4) in MINTREE precise. The first algorithm was favored by Prim and is generally known as the *algorithm of Prim*, although it was already given by Jarník [Jar30].

Algorithm 4.4.3. Let G be a connected graph with vertex set $V = \{1, \ldots, n\}$ given by adjacency lists A_v, and let $w\colon E \to \mathbb{R}$ be a weight function for G.

Procedure PRIM$(G, w; T)$

(1) $g(1) \leftarrow 0$, $S \leftarrow \emptyset$, $T \leftarrow \emptyset$;
(2) **for** $i = 2$ **to** n **do** $g(i) \leftarrow \infty$ **od**
(3) **while** $S \neq V$ **do**
(4) choose $i \in V \setminus S$ such that $g(i)$ is minimal, and set $S \leftarrow S \cup \{i\}$;
(5) **if** $i \neq 1$ **then** $T \leftarrow T \cup \{e(i)\}$ **fi**
(6) **for** $j \in A_i \cap (V \setminus S)$ **do**
(7) **if** $g(v) > w(ij)$ **then** $g(v) \leftarrow w(ij)$ and $e(v) \leftarrow ij$ **fi**
(8) **od**
(9) **od**

Theorem 4.4.4. *Algorithm 4.4.3 determines with complexity $O(|V|^2)$ a minimal spanning tree T for the network (G, w).*

Proof. It is easy to see that Algorithm 4.4.3 is a special case of Algorithm 4.4.1 (written a bit differently): if we always choose V_1 in step (3) of MINTREE, we get the algorithm of Prim. The function $g(i)$ introduced here is just used to simplify finding a shortest edge leaving $V_1 = S$. Hence the algorithm is

correct by Theorem 4.4.2; it remains to discuss its complexity. The **while**-loop is executed $|V|$ times. During each of these iterations, the comparisons in step (4) can be done in at most $|V| - |S|$ steps, so that we get a complexity of $O(|V|^2)$. As G is simple, this is also the overall complexity: in step (6), each edge of G is examined exactly twice. □

Example 4.4.5. Let us apply Algorithm 4.4.3 to the undirected version of the network of Figure 3.5, where we label the edges as follows: $e_1 = \{1,5\}$, $e_2 = \{6,8\}$, $e_3 = \{1,3\}$, $e_4 = \{4,5\}$, $e_5 = \{4,8\}$, $e_6 = \{7,8\}$, $e_7 = \{6,7\}$, $e_8 = \{4,7\}$, $e_9 = \{2,5\}$, $e_{10} = \{2,4\}$, $e_{11} = \{2,6\}$, $e_{12} = \{3,6\}$, $e_{13} = \{5,6\}$, $e_{14} = \{3,8\}$, $e_{15} = \{1,2\}$. Thus the edges are ordered according to their weight. We do not need really this ordering for the algorithm of Prim, but will use it later for the algorithm of Kruskal. The algorithm of Prim then proceeds as summarized in Table 4.2; the resulting minimal spanning tree is indicated by the bold edges in Figure 4.1.

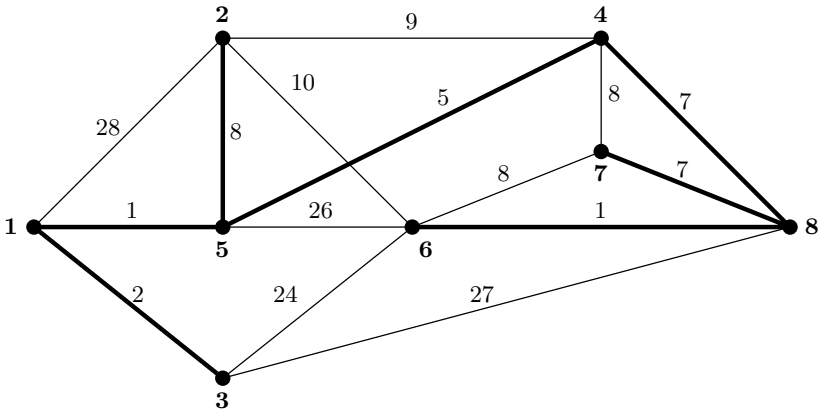


Fig. 4.1. The network specified in Example 4.4.5

Exercise 4.4.6. Let (G, w) be a network. The algorithm of Prim could proceed in many different ways: we might vary the start vertex (which we took to be the vertex 1 in Algorithm 4.4.3), and we might have a choice of the next edge to be added during an iteration (if there are edges of the same minimal weight available). To avoid the latter problem, we simply specify a tiebreaking rule between edges of the same wight (by putting them into a definite order). Show that then always the same minimal spanning tree arises, regardless of the start vertex used [Kra07]. Hint: Use a suitably perturbed weight function and apply Exercise 4.3.5.

Now we turn to the second special case of Algorithm 4.4.1; this is due to Kruskal [Kru56]. We first give a somewhat vague version.

Table 4.2. The algorithm of Prim applied to the network in Figure 4.1

Iteration 1:	$i = 1$, $S = \{1\}$, $T = \emptyset$, $g(2) = 28$, $e(2) = e_{15}$, $g(5) = 1$, $e(5) = e_1$, $g(3) = 2$, $e(3) = e_3$
Iteration 2:	$i = 5$, $S = \{1, 5\}$, $T = \{e_1\}$, $g(2) = 8$, $e(2) = e_9$, $g(4) = 5$, $e(4) = e_4$, $g(6) = 26$, $e(6) = e_{13}$
Iteration 3:	$i = 3$, $S = \{1, 5, 3\}$, $T = \{e_1, e_3\}$, $g(6) = 24$, $e(6) = e_{12}$, $g(8) = 27$, $e(8) = e_{14}$
Iteration 4:	$i = 4$, $S = \{1, 5, 3, 4\}$, $T = \{e_1, e_3, e_4\}$, $g(7) = 8$, $e(7) = e_8$, $g(8) = 7$, $e(8) = e_5$
Iteration 5:	$i = 8$, $S = \{1, 5, 3, 4, 8\}$, $T = \{e_1, e_3, e_4, e_5\}$, $g(6) = 1$, $e(6) = e_2$, $g(7) = 7$, $e(7) = e_6$
Iteration 6:	$i = 6$, $S = \{1, 5, 3, 4, 8, 6\}$, $T = \{e_1, e_3, e_4, e_5, e_2\}$
Iteration 7:	$i = 7$, $S = \{1, 5, 3, 4, 8, 6, 7\}$, $T = \{e_1, e_3, e_4, e_5, e_2, e_6\}$
Iteration 8:	$i = 2$, $S = \{1, 5, 3, 4, 8, 6, 7, 2\}$, $T = \{e_1, e_3, e_4, e_5, e_2, e_6, e_9\}$

Algorithm 4.4.7. Let $G = (V, E)$ be a connected graph with $V = \{1, \ldots, n\}$, and let $w\colon E \to \mathbb{R}$ be a weight function. The edges of G are ordered according to their weight, that is, $E = \{e_1, \ldots, e_m\}$ with $w(e_1) \leq \ldots \leq w(e_m)$.

Procedure KRUSKAL$(G, w; T)$

(1) $T \leftarrow \emptyset$;
(2) **for** $k = 1$ **to** m **do**
(3) **if** e_k does not form a cycle together with some edges of T **then** append e_k to T **fi**
(4) **od**

Note that the algorithm of Kruskal is the special case of MINTREE where V_i and e are chosen in such a way that $w(e)$ is minimal among all edges which are still available: that is, among all those edges which do not have both end vertices in one of the sets V_j and would therefore create a cycle. Again, Theorem 4.4.2 shows that the algorithm is correct. Alternatively, we could also appeal to Theorem 4.3.1 here: in step (3), we choose the edge which does not create a cycle with the edges already in the forest and which has minimal weight among all edges with this property. Thus the set T of edges constructed satisfies (4.1), proving again that T is a minimal spanning tree.

Let us consider the complexity of Algorithm 4.4.7. In order to arrange the edges according to their weight and to remove the edge of smallest weight, we use the data structure *priority queue* already described in Section 3.7. Then these operations can be performed in $O(|E| \log |E|)$ steps. It is more difficult to estimate the complexity of step (3) of the algorithm: how do we check whether an edge creates a cycle, and how many steps does this take?

Here it helps to view the algorithm as a special case of Algorithm 4.4.1. In step (1), we begin with a (totally) disconnected forest T on $n = |V|$ vertices which consists of n trees without any edges. During each iteration, an edge is added to the forest T if and only if its two end vertices are contained in different connected components of the forest constructed so far; these two connected components are then joined by adding the edge to the forest T. Therefore we may check for possible cycles by keeping a list of the connected components; for this task, we need a data structure appropriate for treating partitions. In particular, operations like disjoint unions (MERGE) and finding the component containing a given element should be easy to perform. Using such a data structure, we can write down the following more precise version of Algorithm 4.4.7.

Algorithm 4.4.8. Let $G = (V, E)$ be a connected graph with $V = \{1, \ldots, n\}$, and let $w\colon E \to \mathbb{R}$ be a weight function on G. We assume that E is given as a list of edges.

Procedure KRUSKAL $(G, w; T)$

(1) $T \leftarrow \emptyset$;
(2) **for** $i = 1$ **to** n **do** $V_i \leftarrow \{i\}$ **od**
(3) put E into a priority queue Q with priority function w;
(4) **while** $Q \neq \emptyset$ **do**
(5) $\quad e :=$ DELETEMIN(Q);
(6) $\quad$ find the end vertices u and v of e;
(7) $\quad$ find the components V_u and V_v containing u and v, respectively;
(8) $\quad$ **if** $V_u \neq V_v$ **then** MERGE(V_u, V_v); $T \leftarrow T \cup \{e\}$ **fi**
(9) **od**

Now it is easy to determine the complexity of the iteration. Finding and removing the minimal edge e in the priority queue takes $O(\log |E|)$ steps. Successively merging the original n trivial components and finding the components in step (7) can be done with a total effort of $O(n \log n)$ steps; see [AhHU83] or [CoLR90]. As G is connected, G has at least $n-1$ edges, so that the overall complexity is $O(|E| \log |E|)$. We have established the following result.

Theorem 4.4.9. *The algorithm of Kruskal (as given in 4.4.8) determines with complexity $O(|E| \log |E|)$ a minimal spanning tree for (G, w).* □

For sparse graphs, this complexity is much better than the complexity of the algorithm of Prim. In practice, the algorithm of Kruskal often contains one further step: after each merging of components, it is checked whether there is only one component left; in this case, T is already a tree and we may stop the algorithm.

Example 4.4.10. Let us apply the algorithm of Kruskal to the network of Figure 4.1. The edges $e_1, e_2, e_3, e_4, e_5, e_6$ and e_9 are chosen successively, so that we obtain the same spanning tree as with the algorithm of Prim (although there the edges were chosen in a different order). This has to happen here, since our small example has only one minimal spanning tree. In general, however, the algorithms of Prim and Kruskal will yield different minimal spanning trees.

Exercise 4.4.11. Let (G, w) be a network, and let $\mathfrak{T}$ be the set of all spanning trees of (G, w). Put

$$W(T) \;:=\; \max\{w(e)\colon e \in T\} \qquad \text{for } T \in \mathfrak{T}.$$

Prove that all minimal spanning trees minimize the function W over $\mathfrak{T}$.

Let us turn to our third and final special case of Algorithm 4.4.1; this is due to Boruvka [Bor26a] and requires that all edge weights are distinct. Then we may combine several iterations of MINTREE into one larger step: we always treat each nonempty V_i and add the shortest edge leaving V_i. We shall give a comparatively brief description of the resulting algorithm.

Algorithm 4.4.12. Let $G = (V, E)$ be a connected graph with $V = \{1, \dots, n\}$, and let $w\colon E \to \mathbb{R}$ be a weight function for which two distinct edges always have distinct weights.

Procedure BORUVKA$(G, w; T)$

(1) **for** $i = 1$ **to** n **do** $V_i \leftarrow \{i\}$ **od**
(2) $T \leftarrow \emptyset$; $M \leftarrow \{V_1, \dots, V_n\}$;
(3) **while** $|T| < n - 1$ **do**
(4) **for** $U \in M$ **do**
(5) find an edge $e = uv$ with $u \in U$, $v \notin U$ and $w(e) < w(e')$ for all edges $e' = u'v'$ with $u' \in U$, $v' \notin U$;
(6) find the component U' containing v;
(7) $T \leftarrow T \cup \{e\}$;
(8) **od**
(9) **for** $U \in M$ **do** MERGE(U, U') **od**
(10) **od**

Theorem 4.4.13. *The algorithm of Boruvka determines a minimal spanning tree for (G, w) in $O(|E| \log |V|)$ steps.*

Proof. It follows from Theorem 4.4.2 that the algorithm is correct. The condition that all edge weights are distinct guarantees that no cycles are created during an execution of the **while**-loop. As the number of connected components is at least halved in each iteration, the **while**-loop is executed at most $\log |V|$ times. We leave it to the reader to give a precise formulation of steps (5) and (6) leading to the complexity of $O(|E| \log |V|)$. (Hint: For each vertex v, we should originally have a list E_v of the edges incident with v.) □

Example 4.4.14. Let us apply the algorithm of Boruvka to the network shown in Figure 4.2. When the **while**-loop is executed for the first time, the edges $\{1,2\}$, $\{3,6\}$, $\{4,5\}$, $\{4,7\}$ and $\{7,8\}$ (drawn bold in Figure 4.2) are chosen and inserted into T. That leaves only three connected components, which are merged during the second execution of the **while**-loop by adding the edges $\{2,5\}$ and $\{1,3\}$ (drawn bold broken in Figure 4.2).

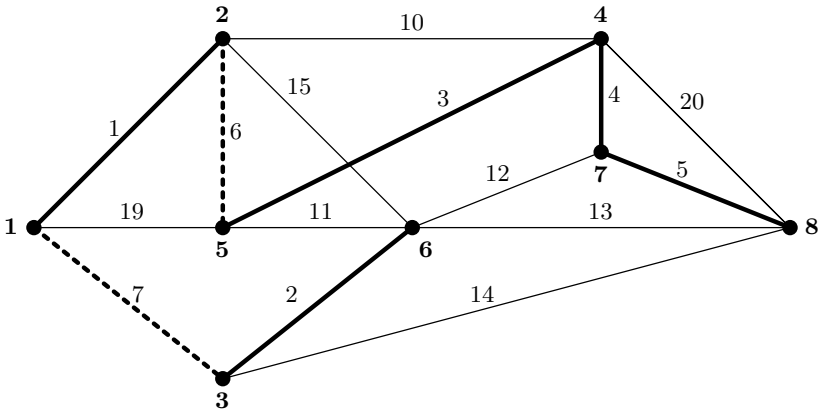


Fig. 4.2. A network

Exercise 4.4.15. Show that the condition that all edge weights are distinct is necessary for the correctness of the algorithm of Boruvka.

Exercise 4.4.16. Table 4.3 gives the distances (in units of 100 miles) between the airports of the cities London, Mexico City, New York, Paris, Peking and Tokyo: Find a minimal spanning tree for the corresponding graph. [BoMu76]

Table 4.3. Distance table for 6 cities

L	MC	NY	Pa	Pe	To	
L	–	56	35	2	51	60
MC	56	–	21	57	78	70
NY	35	21	–	36	68	68
Pa	2	57	36	–	51	61
Pe	51	78	68	51	–	13
To	60	70	68	61	13	–

Exercise 4.4.17. The *tree graph* $T(G)$ of a connected graph G has the spanning trees for G as vertices; two of these trees are adjacent if they have $|V|-2$ edges in common. Prove that $T(G)$ is connected. What can be said about the subgraph of minimal spanning trees (for a given weight function w)?

The complexity of the algorithms discussed in this section can often be improved by using appropriate data structures. Implementations for the algorithms of Prim and Kruskal with complexity $O(|E| \log |V|)$ are given in [Joh75] and [ChTa76]. Using Fibonacci heaps, the algorithm of Prim can be implemented with complexity $O(|E| + |V| \log |V|)$; see [AhMO93]. Boruvka's algorithm (or appropriate variations) can likewise be implemented with complexity $O(|E| \log |V|)$; see [Yao75] and [ChTa76]. Almost linear bounds are in [FrTa87] and [GaGST86]; finally, an algorithm with linear complexity was discovered by Fredman and Willard [FrWi94]; of course, this supposes that the edges are already sorted according to their weights. Unfortunately, the best theoretical algorithms tend to be of no practical interest because of the large size of the implicit constants. There is a simple algorithm with complexity $O(|V|)$ for planar graphs; see [Mat95].

The problem of finding a new minimal spanning tree if we change the weight of an edge and know a minimal spanning tree for the original graph already is discussed in [Fre85] and [Epp94]. On the average, an update may be done in $O(\log |V|)$ steps (under suitable assumptions). Finally, it can be verified in linear time (that is, with complexity $O(|E|)$) whether a given spanning tree is minimal. A similar result holds for the *sensitivity analysis* of minimal spanning trees; this is the problem how much the weight of a given edge e can be increased without changing the minimal spanning tree already known. For the latter two problems, see [DiRT92].

4.5 Maximal spanning trees

For some practical problems, it is necessary to consider *maximal spanning trees*: we want to determine a spanning tree whose weight is maximal among all spanning trees for a given network (G, w). Obviously, a spanning tree T for (G, w) is maximal if and only if T is minimal for $(G, -w)$. Hence we can find a maximal spanning tree by replacing w by $-w$ and using one of the algorithms of Section 4.4. Alternatively, we could also stay with w and just replace *minimum* by *maximum* in the algorithms of Prim, Kruskal and Boruvka; of course, in Kruskal's Algorithm, we then need to order the edges according to decreasing weight.

Let us give some examples where one requires a maximal spanning tree; the first of these is taken from [Chr75].

Example 4.5.1. Consider the problem of sending confidential information to n persons. We define a graph G with n vertices corresponding to the n persons; two vertices i and j are adjacent if it is possible to send information directly from i to j. For each edge ij, let p_{ij} denote the probability that the information sent is overheard; we suppose that these probabilities are independent of each other. Now we replace p_{ij} by $q_{ij} = 1 - p_{ij}$, that is, by the probability that the information is sent without being overheard. In order to send the information

to all n persons, we are looking for a spanning subgraph of G for which the product of the q_{ij} (over the edges occurring in the subgraph) is maximal. Replacing q_{ij} by $w(ij) = \log q_{ij}$, we have reduced our problem to finding a spanning tree of maximal weight.

Problem 4.5.2 (network reliability problem). Let us consider the vertices in Example 4.5.1 as the nodes of a communication network, and let us interpret p_{ij} as the probability that the connection between i and j fails. Then a maximal spanning tree is a tree which maximizes the probability for undisturbed communication between all nodes of the network. This interpretation – and its algorithmic solution – is already contained in [Pri57].

Problem 4.5.3 (bottleneck problem). Let (G, w) be a network, where G is a connected graph, and let

$$W \quad = v_0 \overset{e_1}{\text{---}} v_1 \overset{e_2}{\text{---}} v_2 \ \ldots \ \overset{e_n}{\text{---}} v_n,$$

be any path. Then $c(W) = \min\{w(e_i) : i = 1, \ldots, n\}$ is called the *capacity* or the *inf-section* of W. (We may think of the cross-section of a tube in a supply network or the capacity of a road.) For each pair (u, v) of vertices of G, we want to determine a path from u to v with maximal capacity.

The following theorem due to Hu [Hu61] reduces Problem 4.5.3 to finding a maximal spanning tree. Thus the algorithms of Prim, Kruskal, and Boruvka – modified for determining maximal spanning trees – can be used to solve the bottleneck problem.

Theorem 4.5.4. *Let (G, w) be a network on a connected graph G, and let T be a maximal spanning tree for G. Then, for each pair (u, v) of vertices, the unique path from u to v in T is a path of maximal capacity in G.*

Proof. Let W be the path from u to v in T, and e some edge of W with $c(W) = c(e)$. Suppose there exists a path W' in G having start vertex u and end vertex v such that $c(W') > c(W)$. Let $S_T(e)$ be the cut of G defined in Lemma 4.3.2 and $E(S_T(e))$ the corresponding cocycle. As u and v are in different connected components of $T \setminus e$, the path W' has to contain some edge f of $E(S_T(e))$. As $c(W') > c(W)$, we must have $w(f) > w(e)$. But then $(T \cup \{f\}) \setminus \{e\}$ would be a tree of larger weight than T. □

Exercise 4.5.5. Determine a maximal spanning tree and the maximal capacities for the network of Figure 4.1.

Exercise 4.5.6. Prove the following converse of Theorem 4.5.4. Let T be a spanning tree and assume that, for any two vertices u and v, the unique path from u to v in T is a path of maximal capacity in the network (G, w). Then T is a maximal spanning tree for (G, w).

The following problem is closely related to the bottleneck problem.

Problem 4.5.7 (most uniform spanning tree). Let G be a connected graph and $w\colon E \to \mathbb{R}$ a weight function for G. We ask for a spanning tree T for which the difference between the largest and the smallest edge weights is minimal. This problem can be solved using a modification of the algorithm of Kruskal with complexity $O(|V||E|)$; using a more elaborate data structure, one may even achieve a complexity of $O(|E|\log|V|)$. We refer the reader to [CaMMT85] and [GaSc88].

We remark that analogous problems for digraphs are also of interest. For example, given a digraph having a root, we might want to determine a directed spanning tree of minimal (or maximal) weight. We will return to this problem briefly in Section 4.8.

Exercise 4.5.8. Show that a directed spanning tree of maximal weight in a network (G, w) on a digraph G does not necessarily contain paths of maximal capacity (from the root to all other vertices).

4.6 Steiner trees

Assume that we are faced with the problem of connecting n points in the Euclidean plane by a network of minimal total length; for a concrete example we may think of connecting n cities by a telephone network. Of course, we might just view the given points as the vertices of a complete graph and determine a minimal spanning tree with respect to the Euclidean distance. However, Example 3.2.4 suggests that it should be possible to do better if we are willing to add some new vertices – in our concrete example, we might introduce some switch stations not located in any of the n cities. A plane tree which is allowed to contain – in addition to the n given vertices – an arbitrary number of further vertices, the so-called *Steiner points*, is called a *Steiner tree*. The *euclidean Steiner problem* (called the *geometric Steiner tree problem* in [GaJo76]) is the problem of finding a minimal Steiner tree for the given n vertices.[2]

In the last century Jacob Steiner, among others, studied this problem, which accounts for its name. Actually, the Steiner tree problem for $n = 3$ goes back to Fermat.[3] A fundamental paper on Steiner trees is due to Gilbert and Pollak [GiPo68]; these authors suggested the problem of finding a lower bound

[2] Beware: some authors use the term *Steiner tree* for what we call a *minimal Steiner tree*. As an exercise, the reader might try to settle the geometric Steiner tree problem for the vertices of a unit square: here one gets two Steiner points, and the minimal Steiner tree has length $1+\sqrt{3}$. See [Cox61], Section 1.8, or [CoRo41], p.392.

[3] Here is an exercise for those who remember their high school geometry. Prove that the *Fermat point* of a triangle in which no angle exceeds 120° is the unique point from which the three sides each subtend a 120° angle. See, for example, [Cox61], Section 1.8.

ρ for the ratio between the total length of a minimal Steiner tree and the total length of a minimal spanning tree for a given set of vertices. They were able to show $\rho \geq \frac{1}{2}$ – a result we will prove in Theorem 15.4.9 – and suggested the *Steiner ratio conjecture*: $\rho \geq \sqrt{3}/2$. This bound is optimal, as can be seen rather easily by considering an equilateral triangle; it was finally shown to be correct by Du and Hwang [DuHw90a, DuHw90b]. Thus a minimal Steiner tree for a given set of n vertices is at most (roughly) 14 % better than a minimal spanning tree. We note that minimal Steiner trees are difficult to determine: the euclidean Steiner tree problem is NP-complete, see [GaGJ77]. In contrast, it is easy to find minimal spanning trees. For practical applications, one will therefore be satisfied with minimal spanning trees or with better, but not necessarily minimal, Steiner trees. A relatively good algorithm for determining minimal Steiner trees can be found in [TrHw90]; heuristics for finding good Steiner trees are in [DuZh92].

The Steiner problem has also been studied extensively for other metric spaces. In this section, we consider a graph theoretic version, the *Steiner network problem*. Here one is given a network (G, w) with a positive weight function w, where the vertex set V of G is the disjoint union of two sets R and S. Now a *minimal Steiner tree* is a minimal spanning tree T for an induced subgraph whose vertex set has the form $R \cup S'$ with $S' \subset S$. The vertices in S' are again called *Steiner points*.

Note that the Steiner network problem is a common generalization of two problems for which we have already found efficient solutions: the case $S = \emptyset$ is the problem of determining a minimal spanning tree; and for $|R| = 2$, the problem consists of finding a shortest path between the two given vertices. Nevertheless, the general Steiner network problem is NP-hard, a result due to [Kar72]. [Law76] gave an algorithm whose complexity is polynomial in the cardinality s of S but exponential in the cardinality r of R. Before presenting this algorithm, we prove a further result due to [GiPo68]: one needs only a relatively small number of Steiner points, provided that we are in the metric case, where G is complete and w satisfies the triangle inequality (*metric Steiner network problem*). Then we will show how to reduce the general Steiner network problem to the metric case.

Lemma 4.6.1. *Let $G = (V, E)$ be a complete graph whose vertex set is the disjoint union $V = R \dot\cup S$ of two subsets. Moreover, let w be a positive weight function on E satisfying the triangle inequality. Then there is a minimal Steiner tree for the network (G, w) which contains at most $|R| - 2$ Steiner points.*

Proof. Write $r = |R|$, and let T be a minimal Steiner tree for (G, w) with exactly p Steiner points. Let us denote the average degree of a vertex of R in T by x; similarly, y denotes the average degree of a vertex of S' in T. Then the number of all edges in T satisfies

$$r + p - 1 = \frac{rx + py}{2}.$$

Trivially, $x \geq 1$. As w satisfies the triangle inequality, we may assume that any Steiner point in T is incident with at least three edges, hence $y \geq 3$. This gives $r + p - 1 \geq (r + 3p)/2$; that is, $p \leq r - 2$. □

Lemma 4.6.2. *Let $G = (V, E)$ be a graph whose vertex set is the disjoint union $V = R \dot\cup S$ of two subsets. Moreover, let w be a positive weight function on E and d the distance function in the network (G, w). Then the weight of a minimal Steiner tree for the network (K_V, d) is the same as the weight of a minimal Steiner tree for the original network (G, w).*

Proof. First let T be any Steiner tree for (G, w). Since each edge $e = uv$ of T has weight $w(uv) \geq d(u, v)$, the minimal weight of a Steiner tree for (K_V, d) is at most $w(T)$. Now let us replace each edge uv in a minimal Steiner tree T' for (K_V, d) by the edges of a shortest path from u to v in G. We claim that this yields a Steiner tree T'' of the same weight for (G, w), which will prove the assertion. To justify our claim, we just note that no edge can occur twice and that there cannot be a cycle after replacing the edges, because otherwise we could obtain a Steiner tree from T'' by discarding superfluous edges. As we would have to discard at least one edge, this would give an upper bound $< w(T')$ for the weight of a minimal Steiner tree for (K_V, d) by the first part of our argument, contradicting the minimality of T'. □

Algorithm 4.6.3. Let $G = (V, E)$ be a connected graph with a positive weight function $w\colon E \to \mathbb{R}$, where the vertex set $V = \{1, \ldots, n\}$ is the disjoint union $V = R \dot\cup S$ of two subsets. Write $|R| = r$. The algorithm constructs a minimal Steiner tree T for R in (G, w).

Procedure STEINER$(G, R, w; T)$

```
(1) W ← ∞; T ← ∅; H ← K_n;
(2) FLOYD(G, w; d, p);
(3) for i = 1 to r − 2 do
(4)       for S′ ⊂ S with |S′| = i do
(5)             PRIM(H|(R ∪ S′), d; T′, z);
(6)             if z < W then W ← z; T ← T′ fi
(7)       od
(8) od
(9) for e = uv ∈ T do
(10)      if e ∉ E or w(e) > d(u, v)
(11)      then replace e in T by the edges of a shortest path from
                    u to v in G
(12)      fi
(13) od
```

Here FLOYD is a modified version of the procedure given in Section 3.9 which uses a function p (giving the predecessor as in Algorithm 3.10.1) to determine not only the distance between two vertices, but a shortest path as well. We

need this shortest path in step (11). Similarly, the procedure PRIM is modified in an obvious way to compute not only a minimal spanning tree, but also its weight.

Theorem 4.6.4. *Algorithm 4.6.3 constructs a minimal Steiner tree for* $(G, R; w)$ *with complexity* $O(|V|^3 + 2^{|S|}|R|^2)$.

Proof. In view of Lemma 4.6.1, Lemma 4.6.2 and its proof, and the correctness of the procedures FLOYD and PRIM, Algorithm 4.6.3 is correct. The procedure FLOYD called in step (2) has complexity $O(|V|^3)$ by Theorem 3.9.2. Each call of the procedure PRIM in step (5) has complexity $O(|R|^2)$ by Theorem 4.4.4; note here that PRIM is applied to $O(|R|)$ vertices only, by Lemma 4.6.1. The number of times PRIM is called is obviously

$$\sum_{i=0}^{r-2} \binom{|S|}{i} \leq 2^{|S|}.$$

This establishes the desired complexity bound. □

In particular, Theorem 4.6.4 shows that Algorithm 4.6.3 is polynomial in $|V|$ for fixed s. However, the estimate for the complexity given in the proof of Theorem 4.6.4 is rather bad if we assume r to be fixed; in that case the number of calls of PRIM should better be estimated as about $|S|^{r-2}$. Thus Algorithm 4.6.3 is polynomial for fixed r as well. Altogether, we have proved the following result which generalizes the fact that the Steiner network problem can be solved efficiently for the cases $r = 2$ and $s = 0$, as noted above.

Corollary 4.6.5. *For fixed* r *or for fixed* s *the Steiner network problem can be solved with polynomial complexity.* □

We conclude this section with some recommendations for further reading. A version of the Steiner network problem for digraphs is considered in the survey [Mac87], and an extensive exposition of the various Steiner problems can be found in the book [HwRW92]; more recent books on the subject are [Cie98, Cie01] and [PrSt02]; there is also an interesting collection of articles [DuSR00]. Steiner trees have important applications in VLSI layout; see [KoPS90], [Len90], or [Mar92]. In this context, one is particularly interested in good heuristics; for this topic, we refer to [Vos92], [DuZh92], and [BeRa94]. As this by no means exhaustive collection of references shows, Steiner trees constitute a large and very active area of research.

4.7 Spanning trees with restrictions

In reality, most of the problems one encounters cannot be solved by determining just any (minimal) spanning tree; usually, the solution will have to satisfy

some further restrictions. Unfortunately, this often leads to much harder – quite often even to NP-hard – problems. In this section, we state some of these problems without discussing any possible strategies for solving them (like heuristics); this will be done in Chapter 15 for the TSP as a prototypical example. Even if there is no weight function given, certain restrictions can make the task of finding an appropriate spanning tree NP-hard. The following four problems are all NP-complete; see [GaJo79].

Problem 4.7.1 (degree constrained spanning tree). Let G be a connected graph and k a positive integer. Is there a spanning tree T for G with maximal degree $\Delta \leq k$?

Problem 4.7.2 (maximum leaf spanning tree). Let G be a connected graph and k a positive integer. Is there a spanning tree for G having at least k leaves?

Problem 4.7.3 (shortest total path length spanning tree). Let G be a connected graph and k a positive integer. Is there a spanning tree T such that the sum of all distances $d(u, v)$ over all pairs of vertices $\{u, v\}$ is $\leq k$?

Problem 4.7.4 (isomorphic spanning tree). Let G be a connected graph and T a tree (both defined on n vertices, say). Does G have a spanning tree isomorphic to T? Note that this problem contains the Hamiltonian path problem of Exercise 2.7.7: HP is the special case where T is a path.

We can neither expect to solve these problems efficiently by some algorithm nor to find a nice formula for the value in question – for example, for the maximal number of leaves which a spanning tree of G might have. Nevertheless, it is often still possible to obtain interesting partial results, such as, for example, lower or upper bounds for the respective value. We illustrate this for Problem 4.7.2 and quote a result due to Kleitman and West [KlWe91] which shows that a connected graph with large minimal degree has to contain a spanning tree with many leaves.

Result 4.7.5. *Let $l(n, k)$ be the largest positive integer m such that each connected graph with n vertices and minimal degree k contains a spanning tree with at least m leaves. Then*

(1) $l(n, k) \leq n - 3\frac{n}{\lfloor k-1 \rfloor} + 2$;

(2) $l(n, 3) \geq \frac{n}{4} + 2$;

(3) $l(n, 4) \geq \frac{2n+8}{5}$;

(4) $l(n, k) \geq n\left(1 - \frac{b \ln k}{k}\right)$ *for sufficiently large k, where b is a constant with $b \geq \frac{5}{2}$.* □

We will not include the relatively long (though not really difficult) proof and refer the reader to the original paper instead. The proof given there consists of an explicit construction of a spanning tree with the desired number of leaves.

Now let us turn to some weighted problems with restrictions.

Problem 4.7.6 (bounded diameter spanning tree). Let G be a connected graph with a weight function $w : E \to \mathbb{N}$, and let d and k be two positive integers. Does G contain a spanning tree T with weight $w(T) \leq k$ and diameter at most d?

According to [GaJo79], this problem is NP-complete. Hence it is NP-hard to find among all minimal spanning trees one having the smallest possible diameter. This remains true even if the weight function is restricted to the values 1 and 2 only; however, it is easy to solve the case where all weights are equal.

Exercise 4.7.7. Give a polynomial algorithm for determining a spanning tree whose diameter is at most 1 larger than the smallest possible diameter. Hint: Look at Theorem 3.9.8 and Exercise 4.1.3.

A variation of Problem 4.7.6 was studied in [HoLC91]: one asks for a spanning tree satisfying $w(T) \leq k$ and $d(u, v) \leq d$ for all $u, v \in V$, where $d(u, v)$ is the distance in the network (G, w). This variation is NP-complete as well. However, in a Euclidean graph (that is, the vertices are points in a space $\mathbb{R}^m$ and the weights $w(u, v)$ are given by the Euclidean distance), it is possible to find a spanning tree such that the maximum of the $d(u, v)$ is minimal with complexity $O(|V|^3)$.

Problem 4.7.8 (minimal cost reliability ratio spanning tree). Let G be a connected graph with both a weight function $w\colon E \to \mathbb{N}$ and a reliability function $r\colon E \to (0, 1]$; we interpret $r(e)$ as the probability that edge e works, and $w(e)$ as the cost of using e. Now let T be a spanning tree. As usual, $w(T)$ is the sum of all $w(e)$ with $e \in T$, whereas $r(T)$ is defined to be the product of the $r(e)$ for $e \in T$. Thus $w(T)$ is the total cost of T, and $r(T)$ is the probability that no edge in the tree fails; see Problem 4.5.2. We require a spanning tree T for which the ratio $w(T)/r(T)$ is minimal.

Problem 4.7.8 is one of the few restricted problems for which a polynomial algorithm is known: if we count all arithmetic operations as one step each, it can be solved in $O(|E|^{5/2} \log\log |V|)$ steps;[4] see [ChAN81] and [ChTa84].

Our final example involves two functions on E as well. But this time, the two functions are coupled non-linearly, and our goal is to minimize the resulting function.

[4] As some of the arithmetic operations concerned are exponentiations, this estimate of the complexity might be considered a little optimistic.

Problem 4.7.9 (optimum communication spanning tree). Let G be a connected graph with a weight function $w : E \to \mathbb{N}_0$ and a request function $r : \binom{V}{2} \to \mathbb{N}_0$, and let k be a positive integer. Denote the distance function in the network (T, w) by d; thus $d(u, v)$ is the sum of the weights $w(e)$ of all edges occurring in the unique path from u to v in T. Does G have a spanning tree T satisfying

$$\sum_{\{u,v\} \in \binom{V}{2}} d(u, v) \times r(u, v) \ \leq \ k?$$

In practice, $d(u, v)$ signifies the cost of the path from u to v, and $r(u, v)$ is the capacity we require for communication between u and v – for example, the number of telephone lines needed between cities u and v. Then the product $d(u, v)r(u, v)$ is the cost of communication between u and v, and we want to minimize the total cost.

Problem 4.7.9 is NP-complete even if the request is the same for all edges (*optimum distance spanning tree*); see [JoLR78]. However, the special case where all weights are equal can be solved in polynomial time; see [Hu74] for an algorithm of complexity $O(|V|^4)$. But even this special case of Problem 4.7.9 (*optimum requirement spanning tree*) is much more difficult to solve than the problem of determining a minimal spanning tree , and the solution is found by a completely different method. We shall return to this problem in Section 12.4.

The general problem of finding spanning trees which are optimal with respect to several functions is discussed in [HaRu94].

4.8 Arborescences and directed Euler tours

In this section, we treat the analogue of Theorem 4.2.9 for the directed case and give an application to directed Euler tours. We begin with a simple characterization of arborescences which we used in Section 3.5 already.

Lemma 4.8.1. *Let G be an orientation of a connected graph. Then G is a spanning arborescence with root r if and only if*

$$d_{\text{in}}(v) = 1 \quad \textit{for all } v \neq r \quad \textit{and} \quad d_{\text{in}}(r) = 0. \tag{4.4}$$

Proof. Condition (4.4) is clearly necessary. Thus assume that (4.4) holds. Then G has exactly $|V| - 1$ edges. As $|G|$ is connected by hypothesis, it is a tree by Theorem 1.2.8. Now let v be an arbitrary vertex. Then there is a path W in G from r to v; actually, W is a directed path, as otherwise $d_{\text{in}}(r) \geq 1$ or $d_{\text{in}}(u) \geq 2$ for some vertex $u \neq r$ on W. Thus r is indeed a root for G. □

In analogy to the degree matrix of a graph, we now introduce the *indegree matrix* $D = (d_{ij})_{i,j=1,\ldots,n}$ for a digraph $G = (V, E)$ with vertex set $V = \{1, \ldots, n\}$, where

$$d_{ij} = \begin{cases} d_{\text{in}}(i) & \text{for } i = j \\ -1 & \text{for } ij \in E \\ 0 & \text{otherwise.} \end{cases}$$

We denote the submatrix of D obtained by deleting the i-th row and the i-th column by D_i. The following analogue of Theorem 4.2.9 is due to Tutte [Tut48].

Theorem 4.8.2. *Let $G = (V, E)$ be a digraph with indegree matrix D. Then the r-th minor $\det D_r$ is equal to the number of spanning arborescences of G with root r.*

Proof. We may assume $r = 1$. Note that it is not necessary to consider edges with head 1 if we want to construct spanning arborescences with root 1, and that the entries in the first column of D do not occur in the minor $\det D_1$. Thus we may make the following assumption which simplifies the remainder of the proof considerably: G contains no edges with head 1, and hence the first column of D is the vector having all entries 0. If there should be a vertex $i \neq 1$ with $d_{\text{in}}(i) = 0$, G cannot have any spanning arborescence. On the other hand, the i-th column of D then has all entries equal to 0, so that $\det D_1 = 0$. Thus our assertion is correct for this case, and we may from now on assume that the condition

$$d_{\text{in}}(i) \geq 1 \quad \text{for each vertex } i \neq 1 \tag{4.5}$$

holds. We use induction on $m := d_{\text{in}}(2) + \ldots + d_{\text{in}}(n)$; note $m = |E|$, because of our assumption $d_{\text{in}}(1) = 0$. The more difficult part of the induction here is the induction basis, that is, the case $m = n - 1$. We have to verify that G is an arborescence (with root 1) if and only if $\det D_1 = 1$. First let G be an arborescence; then condition (4.4) holds for $r = 1$. As G is acyclic, G has a topological sorting by Theorem 2.6.3. Thus we may assume $i < j$ for all edges ij in E. Then the matrix D is an upper triangular matrix with diagonal $(0, 1, \ldots, 1)$ and $\det D_1 = 1$.

Conversely, suppose $\det D_1 \neq 0$; we have to show that G is an arborescence (and therefore, actually $\det D_1 = 1$). It follows from condition (4.5) and $m = n - 1$ that $d_{\text{in}}(i) = 1$ for $i = 2, \ldots, n$. Thus G satisfies condition (4.4), and by Lemma 4.8.1 it suffices to show that G is connected. In view of Theorem 1.2.8, we may check instead that G is acyclic. By way of contradiction, suppose that G contains a cycle, say

$$C\colon\ i_1 \text{ — } i_2 \text{ — } \ldots \text{ — } i_k \text{ — } i_1.$$

Let us consider the submatrix U of D_1 which consists of the columns corresponding to $i_1, \ldots, i_k$. As each of the vertices $i_1, \ldots, i_k$ has indegree 1, U can have entries $\neq 0$ only in the rows corresponding to $i_1, \ldots, i_k$. Moreover, the

sum of all rows of U is the zero vector, so that U has rank $\leq k-1$. Thus the columns of U, and hence also the columns of D_1, are linearly dependent; but this implies $\det D_1 = 0$, contradicting our hypothesis. Hence the assertion holds for $m = n-1$.

Now let $m \geq n$. In this case, there has to be a vertex with indegree ≥ 2, say

$$d_{\text{in}}(n) = c \geq 2. \tag{4.6}$$

For each edge e of the form $e = jn$, let $D(e)$ denote the matrix obtained by replacing the last column of D by the vector $\mathbf{v}_e = -\mathbf{e}_j + \mathbf{e}_n$, where $\mathbf{e}_k$ is the k-th unit vector; thus $\mathbf{v}_e$ has entry -1 in row j, entry 1 in row n and all other entries 0. Then $D(e)$ is the indegree matrix for the graph $G(e)$ which arises from G by deleting all edges with head n except for e. Because of (4.6), $G(e)$ has at most $m-1$ edges; hence the induction hypothesis guarantees that the minor $\det D(e)_1$ equals the number of spanning arborescences of $G(e)$ with root 1. Obviously, this is the number of spanning arborescences of G which have root 1 and contain the edge e. Therefore the number of all spanning arborescences of G with root 1 is the sum

$$\det D(e_1)_1 + \ldots + \det D(e_c)_1,$$

where $e_1, \ldots, e_c$ are the c edges of G with head n. On the other hand, the last column of D is the sum $\mathbf{v}_{e_1} + \ldots + \mathbf{v}_{e_c}$ of the last columns of $D(e_1), \ldots, D(e_c)$. Thus the multilinearity of the determinant implies

$$\det D_1 = \det D(e_1)_1 + \ldots + \det D(e_c)_1,$$

and the assertion follows. □

Theorem 4.8.2 can be used to obtain an alternative proof for Theorem 4.2.9. Even though this proof is not shorter than the proof given in Section 4.2, it has the advantage of avoiding the use of the theorem of Cauchy and Binet (which is not all that well-known).

Corollary 4.8.3. *Let $H = (V, E)$ be a graph with adjacency matrix A and degree matrix $D = \operatorname{diag}(\deg 1, \ldots, \deg n) - A$. Then the number of spanning trees of H is the common value of all minors $\det D_r$ of D.*

Proof. Let G be the complete orientation of H. Then there is a one-to-one correspondence between the spanning trees of H and the spanning arborescences of G with root r. Moreover, the degree matrix D of H coincides with the indegree matrix of G. Thus the assertion follows from Theorem 4.8.2. □

Now let G be a directed Eulerian graph; then G is a connected pseudo-symmetric digraph by Theorem 1.6.1. The following theorem of de Bruijn and van Aardenne-Ehrenfest (1951) [deBA51] gives a connection between the spanning arborescences and the directed Euler tours of G.

Theorem 4.8.4. *Let $G = (V, E)$ be an Eulerian digraph. For $i = 1, \ldots, n$, let a_i denote the number of spanning arborescences of G with root i. Then the number e_G of directed Euler tours of G is given by*

$$e_G = a_i \times \prod_{j=1}^{n} (d_{\text{in}}(j) - 1)! \,, \tag{4.7}$$

where i may be chosen arbitrarily.

Sketch of proof. Let A be a spanning arborescence of G with root i. For each vertex $j \neq i$, let e_j denote the unique edge in A with head j, and choose e_i as a fixed edge with head i. Now we construct a cycle C in G by the method described in the algorithm of Hierholzer, using all edges backward (so that we get a directed cycle by reversing the order of the edges in C). That is, we leave vertex i using edge e_i; and, for each vertex j which we reach by using an edge with tail j, we use – as long as this is possible – some edge with head j not yet used to leave j again. In contrast to the algorithm of Hierholzer, we choose e_j for leaving j only after all other edges with head j have been used already. It can be seen as usual that the construction can only terminate at the start vertex i, since G is pseudo-symmetric. Moreover, for each vertex j, all edges with head j – and hence all the edges of G – are used exactly once, because of the restriction that e_j is chosen last. Thus we indeed get an Euler tour. Obviously, whenever we have a choice of an edge in our construction, different choices will give different Euler tours. But the choice of the edges with head j leads to altogether $(d_{\text{in}}(j) - 1)!$ possibilities, so that the product in (4.7) gives the number of distinct Euler tours of G which can be constructed using A. It is easy to see that distinct arborescences with root i also lead to distinct Euler tours. Conversely, we may construct a spanning arborescence with root i from any directed Euler tour in a similar way. □

Corollary 4.8.5. *Let G be an Eulerian digraph. Then the number of spanning arborescences of G with root i is independent of the choice of i.* □

From Exercise 2.3.2 we know that the de Bruijn sequences of length $N = s^n$ over an alphabet S of cardinality s correspond bijectively to the directed Euler tours of the digraph $G_{s,n}$ defined there. Combining Theorems 4.8.2 and 4.8.4, we can now determine the number of such sequences, a result due to de Bruijn [deB46]. See also [vLi74]; a similar method can be found in [Knu67].

Theorem 4.8.6. *The number of de Bruijn sequences of length $N = s^n$ over an alphabet S of cardinality s is*

$$b_{s,n} = s^{-n}(s!)^{s^{n-1}}. \tag{4.8}$$

Sketch of proof. As each vertex of $G_{s,n}$ has indegree s, Theorem 4.8.4 yields

$$b_{s,n} = a((s-1)!)^{s^{n-1}}, \tag{4.9}$$

where a is the common value of all minors of the indegree matrix D of $G_{s,n}$. Thus it remains to show

$$a = s^{s^{n-1}-n}. \tag{4.10}$$

To do this, Theorem 4.8.2 is used. (We have to be a bit careful here, because $G_{s,n}$ contains loops. Of course, these loops should not appear in the matrix D.) As the technical details of calculating the determinant in question are rather tedious, we will not give them here and refer to the literature cited above. □

We conclude this chapter with some references for the problem of determining an arborescence of minimal weight in a network (G, w) on a digraph G. This problem is considerably more difficult than the analogous problem of determining minimal spanning trees in the undirected case; for this reason, we have not treated it in this book. A minimal arborescence can be determined with complexity $O(|V|^2)$ or $O(|E| \log |V|)$; the respective algorithms were found independently by Chu and Liu [ChLi65] and Edmonds [Edm67b]. For an implementation, see [Tar77] and [CaFM79], where some details of Tarjan's paper are corrected, or [GoMi84]. The best result up to now is due in [GaGST86], where Fibonacci heaps are used to achieve a complexity of $O(|V| \log |V| + |E|)$.

5

The Greedy Algorithm

Greed is good. Greed is right. Greed works.
From 'Wall Street'

In this chapter we study a generalization of the algorithm of Kruskal, the so-called *greedy algorithm*. This algorithm can be used for maximization on *independence systems* – in the case of the algorithm of Kruskal, the system of spanning forests of a graph. The greedy strategy is rather short-sighted: we always select the element which seems best at the moment. In other words, among all the admissible elements, we choose one whose weight is maximal and add it to the solution we are constructing. In general, this simple strategy will not work, but for a certain class of structures playing an important part in combinatorial optimization, the so-called *matroids*, it indeed leads to optimal solutions. Actually, matroids may be characterized by the fact that the greedy algorithm works for them, but there are other possible definitions. We will look at various other characterizations of matroids and also consider the notion of matroid duality.

Following this, we shall consider the greedy algorithm as an approximation method for maximization on independence systems which are not matroids. We examine the efficiency of this approach, that is, we derive bounds for the ratio between the solution given by the greedy algorithm and the optimal solution. We also look at the problem of minimization on independence systems. Finally, in the last section, we discuss some further generalizations of matroids and their relationship to the greedy algorithm.

5.1 The greedy algorithm and matroids

Let us begin by recalling the algorithm of Kruskal for determining a maximal spanning tree or forest. Thus let $G = (V, E)$ be a simple graph and $w\colon E \to \mathbb{R}$ a weight function. We order the edges according to decreasing weight and treat them consecutively: an edge is inserted into the set T if and only if it does not form a cycle with the edges which are already contained in T. At the end of the algorithm, T is a maximal spanning forest – or, if G is connected, a maximal spanning tree. We may describe this technique on a slightly more

abstract level as follows. Let $\mathbf{S}$ be the set of all subsets of E which are forests. Then the edge e which is currently examined is added to T if and only if $T \cup \{e\}$ is also in $\mathbf{S}$. Of course, we may apply this strategy – namely choosing the element $e \in E$ which is maximal among all elements of E satisfying a suitable restriction – also to other systems $(E, \mathbf{S})$. We need some definitions.

An *independence system* is a pair $(E, \mathbf{S})$, where E is a set and $\mathbf{S}$ is a subset of the power set of E closed under inclusion: $A \in \mathbf{S}$ and $B \subset A$ imply $B \in \mathbf{S}$. The elements of $\mathbf{S}$ are called *independent sets*. We associate an optimization problem with $(E, \mathbf{S})$ as follows. For a given weight function $w\colon E \to \mathbb{R}_0^+$, we ask for an independent set A with maximal *weight*

$$w(A) := \sum_{e \in A} w(e).$$ [1]

For example, determining a maximal spanning forest for a graph $G = (V, E)$ is the optimization problem associated with $(E, \mathbf{S})$, where $\mathbf{S}$ is the independence system of all edge sets constituting forests. We can now generalize the algorithm of Kruskal to work on an arbitrary independence system.

Algorithm 5.1.1 (greedy algorithm). Let $(E, \mathbf{S})$ be an independence system and $w\colon E \to \mathbb{R}_0^+$ a weight function.

Procedure GREEDY$(E, \mathbf{S}, w; T)$

(1) order the elements of E according to their weight: $E = \{e_1, \ldots, e_m\}$ with $w(e_1) \geq w(e_2) \geq \ldots \geq w(e_m)$;
(2) $T \leftarrow \emptyset$;
(3) **for** $k = 1$ **to** m **do**
(4) **if** $T \cup \{e_k\} \in \mathbf{S}$ **then** append e_k to T **fi**
(5) **od**

By Theorem 4.4.9, the greedy algorithm solves the optimization problem associated with the system of forests of a graph. For arbitrary independence systems, however, the simple strategy – *Always take the biggest piece!* – of this algorithm does not work. We call an independence system $(E, \mathbf{S})$ a *matroid* if the greedy algorithm solves the associated optimization problem correctly.[2] Then we may restate Theorem 4.4.9 as follows.

[1] Note that the restriction to nonnegative weight functions ensures that there is a maximal independent set among the independent sets of maximal weight. We may drop this condition and require A to be a maximal independent set instead; see Theorem 5.5.1.

[2] Originally, Whitney [Whi35] and van der Waerden [vdW37] (see also [vdW49] for an English edition) introduced matroids as an abstract generalization of the notions of linear and algebraic independence, respectively. In the next section, we give some other possible definitions. The generalization of the algorithm of Kruskal to matroids was found independently by [Gal68], [Wel68] and – actually a bit earlier – by Edmonds; see [Edm71]. Early forms of the underlying ideas go back even to [Bor26a] and [Rad57].

Theorem 5.1.2. *Let $G = (V, E)$ be a graph, and let $\mathbf{S}$ be the set of those subsets of E which are forests. Then $(E, \mathbf{S})$ is a matroid.* □

The matroid described above is called the *graphic matroid* of the graph G.[3] Next we treat a class of matroids arising from digraphs.

Theorem 5.1.3. *Let $G = (V, E)$ be a digraph, and let $\mathbf{S}$ be the set of all subsets A of E for which no two edges of A have the same head. Then $(E, \mathbf{S})$ is a matroid, the* head-partition matroid *of G.*[4]

Proof. Obviously, an independent set of maximal weight can be found by choosing, for each vertex v of G with $d_{\text{in}}(v) \neq 0$, the edge with head v having maximal weight. Thus the greedy algorithm solves the corresponding optimization problem. □

Next we give an example where it is absolutely trivial that the greedy algorithm works correctly.

Example 5.1.4. Let E be a set, and let $\mathbf{S}$ be the set of all subsets $X \subseteq E$ with $|X| \leq k$, where $1 \leq k \leq |E|$. Then $(E, \mathbf{S})$ is called a *uniform matroid of degree k*. For $k = |E|$, we also speak of the *free matroid* on E.

Exercise 5.1.5. Let G be a graph. A *matching* in G is a set of edges which do not have any vertices in common; we will study this notion in detail later. Show that the matchings in a graph G do not form a matroid in general, even if G is bipartite. The independence system of matchings in G will be investigated in Section 5.4.

5.2 Characterizations of matroids

We begin with two characterizations of matroids which show that these structures can be viewed as generalizations of the notion of linear independence.

Theorem 5.2.1. *Let $M = (E, \mathbf{S})$ be an independence system. Then the following conditions are equivalent:*

(1) *M is a matroid.*

(2) *For $J, K \in \mathbf{S}$ with $|J| = |K| + 1$, there always exists some $a \in J \setminus K$ such that $K \cup \{a\}$ is also in $\mathbf{S}$.*

(3) *For every subset A of E, all maximal independent subsets of A have the same cardinality.*

---

[3] The construction of graphic matroids admits an obvious generalization to multigraphs. While this observation may, at first glance, not seem very interesting, it will be important in the context of duality.

[4] The *tail-partition matroid* is defined analogously.

Proof. Suppose first that M is a matroid for which (2) is not satisfied. Then there are $J, K \in \mathbf{S}$ with $|J| = |K| + 1$ such that, for every $a \in J \setminus K$, the set $K \cup \{a\}$ is not in $\mathbf{S}$. Let $k = |K|$, and define a weight function w as follows:

$$w(e) := \begin{cases} k+2 & \text{for } e \in K, \\ k+1 & \text{for } e \in J \setminus K, \\ 0 & \text{otherwise.} \end{cases}$$

Note that K is not the solution of the associated optimization problem: $w(K) = k(k+2) < (k+1)^2 \leq w(J)$. On the other hand, the greedy algorithm first chooses all elements of K, because they have maximal weight. Afterwards, the weight of the solution cannot be increased any more: all remaining elements e either have $w(e) = 0$ or are in $J \setminus K$, so that $K \cup \{e\}$ is not in $\mathbf{S}$, according to our assumption above. Thus M is not a matroid, a contradiction. Hence (1) implies (2).

Now let A be an arbitrary subset of E and J and K two maximal independent subsets contained in A; thus there is no independent subset of A containing J or K, except J or K itself, respectively. Suppose we have $|K| < |J|$. As $\mathbf{S}$ is closed under inclusion, there is a subset J' of J with $|J'| = |K| + 1$. By (2), there exists an element $a \in J' \setminus K$ such that $K \cup \{a\}$ is independent, contradicting the maximality of K. Thus (2) implies (3).

Finally, suppose that M is not a matroid, but satisfies condition (3). Then the greedy algorithm does not work for the corresponding optimization problem. Thus we may choose a weight function w for which Algorithm 5.1.1 constructs an independent set $K = \{e_1, \ldots, e_k\}$, even though there exists an independent set $J = \{e'_1, \ldots, e'_h\}$ of larger weight. We may assume that the elements of J and K are ordered according to decreasing weight and that J is a maximal independent subset of E. By construction, K is maximal too. Then (3), with $A = E$, implies $h = k$. We use induction on m to show that the inequality $w(e_i) \geq w(e'_i)$ holds for $i = 1, \ldots, m$; the instance $m = k$ then gives a contradiction to our assumption $w(K) < w(J)$. Now the greedy algorithm chooses e_1 as an element of maximal weight; thus the desired inequality holds for $m = 1$. Now suppose that the assertion holds for $m \geq 1$ and assume $w(e_{m+1}) < w(e'_{m+1})$. Consider the set

$$A \;=\; \{e \in E \colon w(e) \geq w(e'_{m+1})\}.$$

We claim that $S = \{e_1, \ldots, e_m\}$ is a maximal independent subset of A. To see this, let e be any element for which $\{e_1, \ldots, e_m, e\}$ is independent. Then $w(e) \leq w(e_{m+1}) < w(e'_{m+1})$, since the greedy algorithm chose the element e_{m+1} after having chosen e_m; hence $e \notin A$ so that S is indeed a maximal subset of A. But $\{e'_1, \ldots, e'_{m+1}\}$ is also an independent subset of A, contradicting condition (3). Thus (3) implies (1). □

Note that condition (2) of Theorem 5.2.1 is analogous to a well-known result from linear algebra, the Steinitz exchange theorem; therefore (2) is

usually called the *exchange axiom*. Similarly, condition (3) is analogous to the fact that all bases of a linear subspace have the same cardinality. In fact, Theorem 5.2.1 immediately gives the following result.

Theorem 5.2.2. *Let E be a finite subset of a vector space V, and let $\mathbf{S}$ be the set of all linearly independent subsets of E. Then $(E, \mathbf{S})$ is a matroid.* □

A matroid constructed as in Theorem 5.2.2 is called a *vectorial matroid* or a *matric matroid*. The second name comes from the fact that a subset E of a vector space V can be identified with the set of columns of a suitable matrix (after choosing a basis for V); then the independent sets are the linearly independent subsets of this set of columns. An abstract matroid is called *representable over F*, where F is a given field, if it is isomorphic to a vectorial matroid in a vector space V over F. (We leave it to the reader to give a formal definition of the term *isomorphic*.)

Exercise 5.2.3. Prove that every graphic matroid is representable over F for every field F.

Hint: Use the incidence matrix of an arbitrary orientation of the underlying graph.

Exercise 5.2.4. Let $G = (V, E)$ be a graph. A set $A \subseteq E$ is called a *k-forest* of G if it splits into a forest F and at most k edges not in F. Prove that the set of all k-forests of G forms a matroid $M_k(G)$. Hint: Use Theorem 5.2.1 and reduce the assertion to the case $k = 0$, where the matroid in question is just the graphic matroid.

Let us introduce some more terminology chosen in analogy to that used in linear algebra. The maximal independent sets of a matroid $M = (E, \mathbf{S})$ are called its *bases*. The *rank* $\rho(A)$ of a subset A of E is the cardinality of a maximal independent subset of A. Any subset of E not contained in $\mathbf{S}$ is called *dependent*.

Exercise 5.2.5. Let ρ be the rank function of a matroid $M = (E, \mathbf{S})$. Show that ρ has the following properties.

(1) $\rho(A) \leq |A|$ for all $A \subset E$;
(2) ρ is *isotonic*, that is, $A \subset B$ implies $\rho(A) \leq \rho(B)$ (for all $A, B \subset E$);
(3) ρ is *submodular*, that is, $\rho(A \cup B) + \rho(A \cap B) \leq \rho(A) + \rho(B)$ for all $A, B \subset E$.

Conversely, matroids can be defined using their rank function. Let E be a set and ρ a function from the power set of E to $\mathbb{N}_0$ satisfying conditions (1), (2), and (3) above. Then the subsets X of E satisfying $\rho(X) = |X|$ are the independent sets of a matroid on E; for example, see [Wel76]. Submodular functions are important in combinatorial optimization and matroid theory; see, for instance, [PyPe70], [Edm70], [FrTa88], [Qi88], and the monograph by [Fuj91].

To solve Exercise 5.2.5, we need a result worth noting explicitly, although it is a direct consequence of condition (2) of Theorem 5.2.1.

Theorem 5.2.6 (basis completion theorem). *Let J be an independent set of the matroid $M = (E, \mathbf{S})$. Then J is contained in a basis of M.* □

We will now use the rank function to introduce another important concept; this rests on the following simple observation.

Lemma 5.2.7. *Let J be an independent set of the matroid $(E, \mathbf{S})$, and let $X, Y \subseteq E$. If J is a maximal independent set of X as well as of Y, then J is also a maximal independent set of $X \cup Y$.* □

Theorem 5.2.8. *Let $M = (E, \mathbf{S})$ be a matroid and A a subset of E. Then there is a unique maximal set B containing A such that $\rho(A) = \rho(B)$, namely*

$$B \;=\; \{e \in E : \rho(A \cup \{e\}) = \rho(A)\}.$$

Proof. First let C be an arbitrary superset of A satisfying $\rho(A) = \rho(C)$. Then $\rho(A \cup \{e\}) = \rho(A)$ holds for each $e \in C$: otherwise we would have $\rho(C) \geq \rho(A \cup \{e\}) > \rho(A)$. Thus we only need to show that the set B defined in the assertion satisfies the condition $\rho(A) = \rho(B)$. Let J be a maximal independent subset of A; then J is also a maximal independent subset of $A \cup \{e\}$ for each $e \in B$. By Lemma 5.2.7, J is also a maximal independent subset of B. □

The set B defined in Theorem 5.2.8 is called the *span* of A and is denoted by $\sigma(A)$. By analogy with the terminology of linear algebra, a *generating set* of M is a set A with $E = \sigma(A)$. A set A satisfying $\sigma(A) = A$ is called a *closed set*, and a *hyperplane* is a maximal closed proper subset of E. Matroids may be characterized by systems of axioms using the notion of span or of hyperplane; we refer again to [Wel76]. Let us pose some exercises concerning the concepts just introduced.

Exercise 5.2.9. Let $M = (E, \mathbf{S})$ be a matroid. Then the *span operator* σ has the following properties:

(1) $X \subset \sigma(X)$ for all $X \subset E$;
(2) $Y \subset X \Rightarrow \sigma(Y) \subset \sigma(X)$ for all $X, Y \subset E$;
(3) $\sigma(\sigma(X)) = \sigma(X)$ for all $X \subset E$;
(4) If $y \notin \sigma(X)$ and $y \in \sigma(X \cup \{x\})$, then $x \in \sigma(X \cup \{y\})$.

Property (3) explains why the sets $\sigma(A)$ are called *closed*; property (4) is again called the *exchange axiom* because it is basically the same as condition (2) of Theorem 5.2.1. Conversely, the conditions given above can be used for an axiomatic characterization of matroids by the span operator.

Exercise 5.2.10. Show that the bases of a matroid are precisely the minimal generating sets.

Exercise 5.2.11. Let $(E, \mathbf{S})$ be a matroid. Prove the following assertions:
(a) The intersection of closed sets is closed.
(b) $\sigma(X)$ is the intersection of all closed sets containing X.
(c) X is closed if and only if $\rho(X \cup \{x\}) = \rho(X) + 1$ for some $x \in E \setminus X$.

Exercise 5.2.12. Let $(E, \mathbf{S})$ be a matroid of rank r, that is, $\rho(E) = r$). Show that $(E, \mathbf{S})$ contains at least 2^r closed subsets.

Let us introduce one further notion, this time generalizing a concept from graph theory. A *circuit* in a matroid is a minimal dependent set – by analogy with a cycle in a graph. We have the following result; the special case of a graphic matroid should be clear from the preceding discussion.

Theorem 5.2.13. *Let $M = (E, \mathbf{S})$ be a matroid, J an independent set of M, and e any element of $E \setminus J$. Then either $J \cup \{e\}$ is independent, or $J \cup \{e\}$ contains a unique circuit.*

Proof. Suppose that $J \cup \{e\}$ is dependent, and put

$$C \;=\; \{c \in E : (J \cup \{e\}) \setminus \{c\} \in \mathbf{S}\}.$$

Note $C \neq \emptyset$, since $e \in C$ by definition. Also, C is dependent, because otherwise it could be completed to a maximal independent subset K of $J \cup \{e\}$. As J is independent itself, we would have $|K| = |J|$, so that $K = (J \cup \{e\}) \setminus \{d\}$ for some element d. But then d would have to be an element of C, a contradiction. It is easy to see that C is even a circuit: if we remove any element c, we get a subset of $(J \cup \{e\}) \setminus \{c\}$ which is, by definition of C, an independent set. It remains to show that C is the only circuit contained in $J \cup \{e\}$. Thus let D be any circuit contained in $J \cup \{e\}$. Suppose there exists an element $c \in C \setminus D$. Then D is a subset of $(J \cup \{e\}) \setminus \{c\}$ which is an independent set. Therefore $C \subset D$, and hence $C = D$. □

We conclude this section by characterizing matroids in terms of their circuits. We begin with a simple observation.

Lemma 5.2.14. *Let $(E, \mathbf{S})$ be a matroid. A subset A of E is dependent if and only if $\rho(A) < |A|$. Moreover, $\rho(A) = |A| - 1$ for every circuit A.* □

Theorem 5.2.15. *Let $M = (E, \mathbf{S})$ be a matroid, and let $\mathbf{C}$ be the set of all circuits of M. Then $\mathbf{C}$ has the following properties:*

(1) *If $C \subset D$, then $C = D$ for all $C, D \in \mathbf{C}$;*
(2) *For all $C, D \in \mathbf{C}$ with $C \neq D$ and for each $x \in C \cap D$, there always exists some $F \in \mathbf{C}$ with $F \subset (C \cup D) \setminus \{x\}$.*

Conversely, assume that a set system $(E, \mathbf{C})$ satisfies the preceding two circuit axioms. *Then there is a unique matroid $(E, \mathbf{S})$ having $\mathbf{C}$ as its set of circuits.*

Proof. First, let $\mathbf{C}$ be the set of circuits of M. As circuits are minimal dependent sets, condition (1) is trivial. The submodularity of ρ yields, together with Lemma 5.2.14,

$$\rho(C \cup D) + \rho(C \cap D) \leq \rho(C) + \rho(D) = |C| + |D| - 2 = |C \cap D| + |C \cup D| - 2.$$

As C and D are minimal dependent sets, $C \cap D$ is independent; therefore $\rho(C \cap D) = |C \cap D|$, and hence

$$\rho((C \cup D) \setminus \{x\}) \;\leq\; \rho(C \cup D) \;\leq\; |C \cup D| - 2 \;<\; |(C \cup D) \setminus \{x\}|.$$

By Lemma 5.2.14, $(C \cup D) \setminus \{x\}$ is dependent and hence contains a circuit.

Conversely, suppose $\mathbf{C}$ satisfies the conditions (1) and (2). If there exists a matroid $(E, \mathbf{S})$ with set of circuits $\mathbf{C}$, its independent sets are given by

$$\mathbf{S} \;=\; \{J \subset E : J \text{ does not contain any element of } \mathbf{C}\}.$$

Obviously, $\mathbf{S}$ is closed under inclusion, and it suffices to show that $(E, \mathbf{S})$ satisfies condition (2) of Theorem 5.2.1. Suppose that this condition is not satisfied, and choose a counterexample (J, K) such that $|J \cup K|$ is minimal. Let $J \setminus K = \{x_1, \ldots, x_k\}$. Note $k \neq 1$, because otherwise $|J| = |K| + 1$ would imply that K is a subset of J, and hence $J = K \cup \{x_1\}$ would be independent. Our assumption means $K \cup \{x_i\} \notin \mathbf{S}$ for $i = 1, \ldots, k$. In particular, there exists $C \in \mathbf{C}$ with $C \subset K \cup \{x_1\}$; as K is independent, x_1 must be in C. As J is independent, there is an element $y \in K \setminus J$ which is contained in C. Consider the set $Z = (K \setminus \{y\}) \cup \{x_1\}$. If Z is not in $\mathbf{S}$, then there exists $D \in \mathbf{C}$ with $D \subset Z$ and $x_1 \in D$, and the circuit axiom (2) yields a set $F \in \mathbf{C}$ with $F \subset (C \cup D) \setminus \{x_1\} \subset K$, contradicting $K \in \mathbf{S}$. Hence Z must be independent. Note that $|Z \cup J| < |K \cup J|$. As we chose our counterexample (J, K) to be minimal, (J, Z) has to satisfy condition (2) of Theorem 5.2.1. Thus there exists some x_i, say x_2, such that $Z \cup \{x_2\} \in \mathbf{S}$. But $K \cup \{x_2\} \notin \mathbf{S}$, so that there is a circuit $C' \in \mathbf{C}$ with $C' \subset K \cup \{x_2\}$. We must have $x_2 \in C'$, because K is independent; and $(K \setminus \{y\}) \cup \{x_1, x_2\} \in \mathbf{S}$ yields $y \in C'$. Thus $C' \neq C$, and $y \in C \cap C'$. Using the circuit axiom (2) again, there exists a set $F' \in \mathbf{C}$ with $F' \subset (C \cup C') \setminus \{y\} \subset (K \setminus \{y\}) \cup \{x_1, x_2\} \in \mathbf{S}$. This contradicts the definition of $\mathbf{S}$. Therefore $M = (E, \mathbf{S})$ is indeed a matroid, and clearly $\mathbf{C}$ is the set of circuits of M. □

Exercise 5.2.16. Show that the set $\mathbf{C}$ of circuits of a matroid $(E, \mathbf{S})$ actually satisfies the following stronger version of the circuit axiom (2) in Theorem 5.2.15 [Leh64]:

(2′) For all $C, D \in \mathbf{C}$, for each $x \in C \cap D$, and for each $y \in C \setminus D$, there exists a set $F \in \mathbf{C}$ with $y \in F \subset (C \cup D) \setminus \{x\}$.

5.3 Matroid duality

In this section we construct the *dual matroid* M^* of a given matroid M. We stress that the notion of duality of matroids differs from the duality known from linear algebra: the dual matroid of a finite vector space is *not* the matroid formed by the dual space. Matroid duality has an interesting meaning in graph theory; see Result 5.3.4 below. The following construction of the dual matroid is due to Whitney [Whi35].

Theorem 5.3.1. *Let $M = (E, \mathbf{S})$ be a matroid. Put $M^* = (E, \mathbf{S}^*)$, where*

$$\mathbf{S}^* \;=\; \{J \subset E\colon\; J \subset E \setminus B \text{ for some basis } B \text{ of } M\}.$$

Then M^* is a matroid as well, and the rank function ρ^* of M^* is given by

$$\rho^*(A) \;=\; |A| + \rho(E \setminus A) - \rho(E). \tag{5.1}$$

Proof. Obviously, $\mathbf{S}^*$ is closed under inclusion. By Theorem 5.2.1, it suffices to verify the following condition for each subset A of E: all maximal subsets of A which are independent with respect to $\mathbf{S}^*$ have the cardinality $\rho^*(A)$ given in (5.1). Thus let J be such a subset of A. Then there exists a basis B of M with $J = (E \setminus B) \cap A$; moreover, J is maximal with respect to this property. This means that B is chosen such that $A \setminus J = A \setminus ((E \setminus B) \cap A) = A \cap B$ is minimal with respect to inclusion. Hence $K := (E \setminus A) \cap B$ is maximal with respect to inclusion. Thus K is a basis of $E \setminus A$ in the matroid M and has cardinality $\rho(E \setminus A)$. Therefore the minimal subsets $A \cap B$ all have cardinality

$$|B| - |K| \;=\; |B| - \rho(E \setminus A) \;=\; \rho(E) - \rho(E \setminus A);$$

and all maximal subsets $J \in \mathbf{S}^*$ of A have cardinality

$$|J| \;=\; |A| - |A \setminus J| \;=\; |A| - |A \cap B| \;=\; |A| + \rho(E \setminus A) - \rho(E). \qquad \square$$

The matroid M^* constructed in Theorem 5.3.1 is called the *dual matroid* of M. The bases of M^* are the *cobases* of M; the circuits of M^* are the *cocircuits* of M. According to Exercise 5.2.10, the independent sets of M^* are precisely the complements of generating sets of M. This implies the following result.

Corollary 5.3.2. *Let $M = (E, \mathbf{S})$ be a matroid. Then the independent sets of M^* are the complements of the generating sets of M. In particular, the bases of M^* are the complements of the bases of M. Hence $(M^*)^* = M$.* $\square$

Example 5.3.3. Let $M = M(G)$ be the matroid corresponding to a connected graph G. Then the bases of M are the spanning trees of G, and the bases of M^* are the *cotrees*, that is, the complements of the spanning trees. More generally, a set S is independent in M^* if and only if its complement

$\overline{S}$ contains a spanning tree of G, that is, if and only if $\overline{S}$ is connected. By definition, the circuits of a matroid are the minimal dependent sets. Thus the circuits of M are the cycles in G, and the circuits of M^* are the minimal sets C for which the complement $\overline{C}$ is not connected. In other words, the circuits of M^* are the *simple cocycles* of G – all those cocycles which are minimal with respect to inclusion.

In the general case, if G has p connected components, n vertices, and m edges, then $M(G)$ has rank $n-p$ and $M(G)^*$ has rank $m-(n-p)$, by Theorem 4.2.4.

We now state an important theorem due to Whitney [Whi33] which clarifies the role of matroid duality in graph theory; a proof can be found in [Wel76] or [Oxl92].

Result 5.3.4. *A graph G is planar if and only if the dual matroid $M(G)^*$ is graphic.* □

Remark 5.3.5. While a proof of Theorem 5.3.4 is beyond the scope of this book, let us at least give a rough idea how the dual matroid of a planar graph G can be seen to be graphic; to simplify matters, we shall assume that each edge lies in a cycle. Suppose G is drawn in the plane. Construct a multigraph $G^* = (V^*, E^*)$ whose vertices correspond to the faces of G, by selecting a point v_F in the interior of each face F; two such points are connected by as many edges as the corresponding faces share in G.

More precisely, assume that the boundaries of two faces F and F' share exactly k edges, say $e_1, \ldots, e_k$. Then the corresponding vertices v_F and $v_{F'}$ are joined by k edges $e'_1, \ldots, e'_k$ drawn in such a way that the edge e'_i crosses the edge e_i but no other edge of the given drawing of G. This results in a plane multigraph G^*, and one may show $M(G)^* \cong M(G^*)$. The planar multigraph G^* is usually called a *geometric dual* of G.[5] See Figure 5.1 for an example of the construction just described, using $G = K_5 \setminus e$, which was shown to be planar in Exercise 1.5.6; here we actually obtain a graph. The reader might find it instructive to draw a few more examples, for instance using $G = K_{3,3} \setminus e$ (where a multigraph arises).

Exercise 5.3.6. Let $M = (E, \mathbf{S})$ be a matroid, and let A and A^* be two disjoint subsets of E. If A is independent in M and if A^* is independent in M^*, then there are bases B and B^* of M and M^*, respectively, with $A \subset B$, $A^* \subset B^*$, and $B \cap B^* = \emptyset$. Hint: Note $\rho(E) = \rho(E \setminus A^*)$.

Exercise 5.3.7. Let $M = (E, \mathbf{S})$ be a matroid. A subset X of E is a basis of M if and only if X has nonempty intersection with each cocircuit of M and is minimal with respect to this property.

[5] If edges not lying in a cycle – that is, edges belonging to just one face F – are allowed, one has to associate such edges with loops in the construction of G^*. It should also be noted that a planar graph may admit essentially different plane embeddings and, hence, nonisomorphic geometric duals.

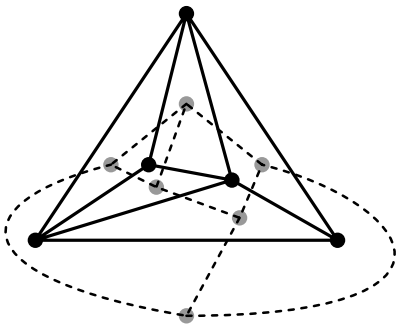


Fig. 5.1. A geometric dual of $K_5 \setminus e$

Exercise 5.3.8. Let C be a circuit and C^* a cocircuit of the matroid M. Prove $|C \cap C^*| \neq 1$. Hint: Use Exercise 5.3.6 for an indirect proof.

This result plays an important role in characterizing a pair (M, M^*) of dual matroids by the properties of their circuits and cocircuits; see [Min66].

Exercise 5.3.9. Let x and y be two distinct elements of a circuit C in a matroid M. Then there exists a cocircuit C^* in M such that $C \cap C^* = \{x, y\}$. Hint: Complete $C \setminus \{x\}$ to a basis B of M and consider $B^* \cup \{y\}$, where $B^* = E \setminus B$ is a cobasis.

We will return to matroids several times throughout this book. For a thorough study of matroid theory we recommend the book by Welsh [Wel76], which is still a standard reference. We also mention the monographs [Tut71], [Rec89], and [Oxl92]; of these, Oxley's book is of particular interest as it also includes applications of matroids. A series of monographs concerning matroid theory was edited by White [Whi86, Whi87, Whi92].

5.4 The greedy algorithm as an approximation method

In this section we investigate independence systems $M = (E, \mathbf{S})$ which are not matroids. By definition, the greedy algorithm then does not (always) yield an optimal solution for the optimization problem

$$\text{(P)} \qquad \text{determine } \max\{w(A) : A \in \mathbf{S}\},$$

where $w : E \to \mathbb{R}_0^+$ is a given weight function. Of course, we may apply the greedy algorithm nevertheless, in the hope of obtaining a reasonably good approximate solution in this way. We shall examine the quality of this approach by deriving bounds for the term

$$f(M) \;=\; \min\left\{\frac{w(T_g)}{w(T_0)} :\; w\colon E \to \mathbb{R}_0^+\right\},$$

where T_g denotes a solution for (P) constructed by the greedy algorithm, whereas T_0 is an optimal solution.[6] We follow Korte and Hausmann [KoHa78] in this section; similar results were also obtained by Jenkyns [Jen76].

First we introduce some useful parameters for independence systems. For any subset A of E, the *lower rank* of A is

$$\mathrm{lr}(A) \;=\; \min\{|I| :\; I \subset A,\, I \in \mathbf{S},\, I \cup \{a\} \notin \mathbf{S} \text{ for all } a \in A \setminus I\}.$$

Similarly, we define the *upper rank* of A as

$$\mathrm{ur}(A) \;=\; \max\{|I| :\; I \subset A,\, I \in \mathbf{S},\, I \cup \{a\} \notin \mathbf{S} \text{ for all } a \in A \setminus I\}.$$

Moreover, the *rank quotient* of M is

$$\mathrm{rq}(M) \;=\; \min\left\{\frac{\mathrm{lr}(A)}{\mathrm{ur}(A)} :\; A \subset E\right\};$$

here terms $\frac{0}{0}$ might occur; such terms are considered to have value 1. Note that Theorem 5.2.1 immediately yields the following result.

Lemma 5.4.1. *An independence system $M = (E, \mathbf{S})$ is a matroid if and only if* $\mathrm{rq}(M) = 1$. □

As we will see, the rank quotient indicates how much M differs from a matroid. Below, we will get an interesting estimate for the rank quotient confirming this interpretation. But first, we prove the following theorem of [Jen76] and [KoHa78][7], which shows how the quality of the solution found by the greedy algorithm depends on the rank quotient of M.

Theorem 5.4.2. *Let $M = (E, \mathbf{S})$ be an independence system with a weight function $w\colon E \to \mathbb{R}_0^+$. Moreover, let T_g be a solution of problem (P) found by the greedy algorithm, and T_0 an optimal solution. Then*

$$\mathrm{rq}(M) \;\le\; \frac{w(T_g)}{w(T_0)} \;\le\; 1.$$

Proof. The second inequality is trivial. To prove the first inequality, we introduce the following notation. Suppose the set E is ordered according to decreasing weight, say $E = \{e_1, \ldots, e_m\}$ with $w(e_1) \ge w(e_2) \ge \ldots \ge w(e_m)$. We put $w(e_{m+1}) = 0$ and write

[6] Note that the greedy algorithm might yield different solutions T_g for different orderings of the elements of E (which may occur if there are distinct elements having the same weight). Hence we also have to minimize over all T_g.

[7] This result was conjectured or even proved somewhat earlier by various other authors; see the remarks in [KoHa78].

$$E_i \;=\; \{e_1, \ldots, e_i\} \quad \text{for } i = 1, \ldots, m.$$

Then we get the following formulae:

$$w(T_g) \;=\; \sum_{i=1}^{m} |T_g \cap E_i|(w(e_i) - w(e_{i+1})); \tag{5.2}$$

$$w(T_0) \;=\; \sum_{i=1}^{m} |T_0 \cap E_i|(w(e_i) - w(e_{i+1})). \tag{5.3}$$

Now $T_0 \cap E_i$ is an independent subset of E_i, and thus $|T_0 \cap E_i| \leq \mathrm{ur}(E_i)$. By definition of the greedy algorithm, $T_g \cap E_i$ is a maximal independent subset of E_i, and therefore $|T_g \cap E_i| \geq \mathrm{lr}(E_i)$. Using these two observations, we obtain

$$|T_g \cap E_i| \;\geq\; |T_0 \cap E_i| \times \frac{\mathrm{lr}(E_i)}{\mathrm{ur}(E_i)} \;\geq\; |T_0 \cap E_i| \times \mathrm{rq}(M).$$

Using (5.2) and (5.3) yields

$$\begin{aligned} w(T_g) \;&=\; \sum_{i=1}^{m} |T_g \cap E_i|(w(e_i) - w(e_{i+1})) \\ &\geq\; \mathrm{rq}(M) \times \sum_{i=1}^{m} |T_0 \cap E_i|(w(e_i) - w(e_{i+1})) \\ &=\; \mathrm{rq}(M) \times w(T_0). \qquad \square \end{aligned}$$

As w and T_g were chosen arbitrarily in Theorem 5.4.2, we conclude $\mathrm{rq}(M) \leq f(M)$. The following result shows that we actually have equality.

Theorem 5.4.3. *Let $M = (E, \mathbf{S})$ be an independence system. Then there exist a weight function $w\colon E \to \mathbb{R}_0^+$ and a solution T_g for problem (P) obtained by the greedy algorithm such that*

$$\frac{w(T_g)}{w(T_0)} \;=\; \mathrm{rq}(M),$$

where T_0 denotes an optimal solution for (P).

Proof. Choose a subset A of E with $\mathrm{rq}(M) = \mathrm{lr}(A)/\mathrm{ur}(A)$, and let I_l and I_u be maximal independent subsets of A satisfying $|I_l| = \mathrm{lr}(A)$ and $|I_u| = \mathrm{ur}(A)$. Define the weight function w by

$$w(e) := \begin{cases} 1 & \text{for } e \in A \\ 0 & \text{otherwise} \end{cases}$$

and order the elements $e_1, \ldots, e_m$ of E such that

$$I_l = \{e_1, \ldots, e_{\mathrm{lr}(A)}\}, \;\; A \setminus I_l = \{e_{\mathrm{lr}(A)+1}, \ldots, e_{|A|}\}, \;\; E \setminus A = \{e_{|A|+1}, \ldots, e_m\}.$$

Then I_l is the solution for (P) found by the greedy algorithm with respect to this ordering of the elements of E, whereas I_u is an optimal solution. Hence

$$\frac{w(I_l)}{w(I_u)} \;=\; \frac{|I_l|}{|I_u|} \;=\; \frac{\mathrm{lr}(A)}{\mathrm{ur}(A)} \;=\; \mathrm{rq}(M). \qquad \square$$

As Theorems 5.4.2 and 5.4.3 show, the rank quotient of an independence system gives a tight bound for the weight of the greedy solution in comparison to the optimal solution of (P); thus we have obtained the desired measure for the quality of the greedy algorithm as an approximation method. Of course, this leaves us with the nontrivial problem of determining the rank quotient for a given independence system M. The following result provides an example where it is possible to determine this invariant explicitly.

Theorem 5.4.4. *Let $G = (V, E)$ be a graph and $M = (E, \mathbf{S})$ the independence system given by the set of all matchings of G; see Exercise 5.1.5. Then $\mathrm{rq}(M) = 1$ provided that each connected component of G is isomorphic either to a complete graph K_i with $i \leq 3$ or to a star. In all other cases, $\mathrm{rq}(M) = \frac{1}{2}$.*

Proof. First we prove $\mathrm{rq}(M) \geq \frac{1}{2}$. Thus we need to show

$$\frac{\mathrm{lr}(A)}{\mathrm{ur}(A)} \geq \frac{1}{2} \qquad \text{for all } A \subset E.$$

Let I_1 and I_2 be two maximal independent subsets of A, that is, two maximal matchings contained in A. Obviously, it suffices to show $|I_1| \geq \frac{1}{2}|I_2|$. We define a mapping $\alpha : I_2 \setminus I_1 \to I_1 \setminus I_2$ as follows. Let e be any edge in $I_2 \setminus I_1$. As $I_1 \cup \{e\} \subset A$ and as I_1 is a maximal independent subset of A, $I_1 \cup \{e\}$ cannot be a matching. Thus there exists an edge $\alpha(e) \in I_1$ which has a vertex in common with e. As I_2 is a matching, we cannot have $\alpha(e) \in I_2$, so that we have indeed defined a mapping $\alpha : I_2 \setminus I_1 \to I_1 \setminus I_2$. Clearly, each edge $e \in I_1 \setminus I_2$ can share a vertex with at most two edges in $I_2 \setminus I_1$, so that e has at most two preimages under α. Therefore

$$|I_1 \setminus I_2| \;\geq\; |\alpha(I_2 \setminus I_1)| \;\geq\; \frac{1}{2}|I_2 \setminus I_1|$$

and hence

$$|I_1| \;=\; |I_1 \setminus I_2| + |I_1 \cap I_2| \;\geq\; \frac{1}{2}|I_2 \setminus I_1| + \frac{1}{2}|I_1 \cap I_2| \;=\; \frac{1}{2}|I_2|.$$

Assume first that each connected component of G is isomorphic either to a complete graph K_i with $i \leq 3$ or to a star. Then $\mathrm{lr}(A) = \mathrm{ur}(A)$ for all $A \subset E$, so that $\mathrm{rq}(M) = 1$. In all other cases, G contains a subgraph (U, A) isomorphic to a path P_3 of length 3. Then $\mathrm{lr}(A) = 1$ and $\mathrm{ur}(A) = 2$, so that $\mathrm{rq}(M) \leq \frac{1}{2}$. Together with the converse inequality already proved, this establishes the theorem. $\square$

Let us mention a further similar result without proof; the interested reader is referred to [KoHa78].

Result 5.4.5. *Let $G = (V, E)$ be the complete graph on n vertices, and let $M = (E, \mathbf{S})$ be the independence system whose independent sets are the subsets of Hamiltonian cycles of G. Then*

$$\frac{1}{2} \;\leq\; \mathrm{rq}(M) \;\leq\; \frac{1}{2} + \frac{3}{2n}. \qquad \square$$

By definition, the maximal independent sets of an independence system as in Result 5.4.5 are precisely the Hamiltonian cycles of G. Thus the greedy algorithm provides a good approximation to the optimal solution of the problem of finding a Hamiltonian cycle of maximal weight in K_n for a given weight function $w : E \to \mathbb{R}_0^+$. Note that this is the problem opposite to the TSP, where we look for a Hamiltonian cycle of minimal weight.

The above examples suggest that the greedy algorithm can be a really good approximation method. Unfortunately, this is not true in general. As the following exercise shows, it is easy to construct an infinite class of independence systems whose rank quotient becomes arbitrarily small.

Exercise 5.4.6. Let G be the complete digraph on n vertices, and let $M = (E, \mathbf{S})$ be the independence system determined by the acyclic directed subgraphs of G, that is,

$$\mathbf{S} \;=\; \{D \subset E : \, D \text{ does not contain any directed cycle}\}.$$

Then $\mathrm{rq}(M) \leq 2/n$, so that $\lim\limits_{n\to\infty} \mathrm{rq}(M) = 0$.

Our next aim is to derive a useful estimate for the rank quotient of an independence system. We need a lemma first.

Lemma 5.4.7. *Every independence system $M = (E, \mathbf{S})$ can be represented as the intersection of finitely many matroids on E.*

Proof. We have to show the existence of a positive integer k and matroids $M_1 = (E, \mathbf{S}_1), \ldots, M_k = (E, \mathbf{S}_k)$ satisfying $\mathbf{S} = \bigcap_{i=1}^{k} \mathbf{S}_i$. Let $C_1, \ldots, C_k$ be the minimal elements of the set $\{A \subset E : A \notin \mathbf{S}\}$, that is, the circuits of the independence system M. It is easily seen that

$$\mathbf{S} \;=\; \bigcap_{i=1}^{k} \mathbf{S}_i, \quad \text{where } \mathbf{S}_i := \{A \subseteq E : \, C_i \not\subseteq A\}.$$

Thus we want to show that all $M_i = (E, \mathbf{S}_i)$ are matroids. So let A be an arbitrary subset of E. If C_i is not a subset of A, then A is independent in M_i, so that A itself is the only maximal independent subset of A in M_i. Now suppose $C_i \subseteq A$. Then, by definition, the maximal independent subsets of A in M_i are the sets of the form $A \setminus \{e\}$ for some $e \in C_i$. Thus all maximal independent subsets of A have the same cardinality $|A| - 1$ in this case. This shows that M_i satisfies condition (3) of Theorem 5.2.1, so that M_i is indeed a matroid. $\square$

Theorem 5.4.8. *Let the independence system $M = (E, \mathbf{S})$ be the intersection of k matroids $M_1, \ldots, M_k$ on E. Then* $\mathrm{rq}(M) \geq 1/k$.

Proof. Let A be any subset of E, and let I_1, I_2 be any two maximal independent subsets of A. Obviously, it suffices to show $k|I_1| \geq |I_2|$. For $i = 1, \ldots, k$ and $j = 1, 2$, let $I_{i,j}$ be a maximal independent subset of $I_1 \cup I_2$ containing I_j (in M_i). Suppose there exists an element $e \in I_2 \setminus I_1$ with $e \in I_{i,1} \setminus I_1$ for $i = 1, \ldots, k$. Then

$$I_1 \cup \{e\} \subseteq \bigcap_{i=1}^{k} I_{i,1} \in \mathbf{S},$$

contradicting the maximality of I_1. Hence each $e \in I_2 \setminus I_1$ can be contained in at most $k - 1$ of the sets $I_{i,1} \setminus I_1$; this implies

$$(*) \quad \sum_{i=1}^{k} |I_{i,1}| - k|I_1| = \sum_{i=1}^{k} |I_{i,1} \setminus I_1| \leq (k-1)|I_2 \setminus I_1| \leq (k-1)|I_2|.$$

As all the M_i are matroids, we have $|I_{i,1}| = |I_{i,2}|$ for $i = 1, \ldots, k$ and hence, using $(*)$,

$$|I_2| \leq |I_2| + \sum_{i=1}^{k} |I_{i,2} \setminus I_2| = \sum_{i=1}^{k} |I_{i,2}| - (k-1)|I_2|$$

$$= \sum_{i=1}^{k} |I_{i,1}| - (k-1)|I_2| \leq k|I_1|. \quad \square$$

For each positive integer k, there exists an independence system for which the bound of Theorem 5.4.8 is tight. Unfortunately, equality does not hold in general; for instance, Result 5.4.5 provides a family of counterexamples. The interested reader is referred to [KoHa78].

Example 5.4.9. Let $G = (V, E)$ be a strongly connected digraph, and let M be the intersection of the graphic matroid and the head-partition matroid of G. Then the independent sets of maximal cardinality in M are precisely the spanning arborescences of G. Note that M may admit further maximal independent sets, as an arbitrary arborescence does not necessarily extend to a spanning arborescence: in general, M is not a matroid.

Exercise 5.4.10. Let G be a digraph. Find three matroids such that each directed Hamiltonian path in G is an independent set of maximal cardinality in the intersection of these matroids.

In the situation of Example 5.4.9, the greedy algorithm constructs an arborescence whose weight is at least half of the weight of a maximal arborescence, by Theorems 5.4.8 and 5.4.2. As mentioned at the end of Section 4.8, a maximal arborescence can be found with complexity $O(|E| \log |V|)$, using a considerably more involved method. The following result about the intersection of two arbitrary matroids is interesting in this context.

Result 5.4.11. *Consider an independence system $M = (E, \mathbf{S})$ which is the intersection of two matroids $M_1 = (E, \mathbf{S}_1)$ and $M_2 = (E, \mathbf{S}_2)$, and let $w : E \to \mathbb{R}_0^+$ be a weight function on M. Assume that we may check in polynomial time whether a set is independent in either M_1 or M_2. Then it is also possible to find an independent set of maximal weight in M in polynomial time.* □

For a situation as described in Result 5.4.11, we say that the two matroids M_1 and M_2 are given by *oracles for independence*; this just means that it is somehow possible to check whether a given set is independent in polynomial time. Then Result 5.4.11 states that a maximal independent set in M can be found in *oracle polynomial time*, that is, by using both oracles a polynomial number of times; see [HaKo81] for more on oracles representing matroids and independence systems. Result 5.4.11 is very important in combinatorial optimization. We have decided to omit the proof because the corresponding algorithms as well as the proofs for correctness are rather difficult – even in the case without weights – and use tools from matroid theory which go beyond the limits of this book. The interested reader may consult [Law75], [Edm79], and [Cun86]; or the books [Law76] and [Whi87].

Of course, one may also consider the analogous problems for the intersection of three or more matroids; we will just state the version without weights. Unfortunately, these problems are presumably not solvable in polynomial time, as the next result indicates.

Problem 5.4.12 (matroid intersection problem, MIP). Let three matroids $M_i = (E, \mathbf{S}_i)$, $i = 1, 2, 3$, be given, and let k be a positive integer. Does there exist a subset A of E with $|A| \geq k$ and $A \in \mathbf{S}_1 \cap \mathbf{S}_2 \cap \mathbf{S}_3$?

Theorem 5.4.13. *MIP is NP-complete.*

Proof. Exercise 5.4.10 shows that the question whether a given digraph contains a directed Hamiltonian path is a special case of MIP. This problem (*directed Hamiltonian path*, DHP) is NP-complete, as the analogous problem HP for the undirected case is NP-complete by Exercise 2.7.7, and as HP can be transformed polynomially to DHP by replacing a given graph by its complete orientation. Hence the more general MIP is NP-complete, too. □

Theorem 5.4.13 indicates that the results presented in this chapter really are quite remarkable: even though the problem of determining a maximal independent set in the intersection of $k \geq 3$ matroids is NP-hard (maximal either with respect to cardinality or a more general weight function), the greedy algorithm gives a quite simple polynomial method for finding an approximate solution which differs at most by a fixed ratio from the optimal solution. This result is by no means trivial, as there are many optimization problems for which even the question whether an approximate solution with a performance guaranty exists is NP-hard; we will encounter an example for this phenomenon in Chapter 15.

5.5 Minimization in independence systems

In this section we consider the minimization problem for independence systems, that is, the problem of finding a maximal independent set of minimal weight; this turns out to be easy for matroids. We first show that the greedy algorithm actually works for arbitrary weight functions on a matroid.

Theorem 5.5.1. *Let $M = (E, \mathbf{S})$ be a matroid, and let $w : E \to \mathbb{R}$ be any weight function on M. Then the greedy algorithm finds an optimal solution for the problem*

$$\text{(BMAX)} \qquad \textit{determine} \quad \max\{w(B) \colon B \textit{ is a basis of } M\}.$$

Proof. By definition, the assertion holds if all weights are nonnegative. Otherwise, we put

$$C \;=\; \max\{-w(e) : e \in E,\, w(e) < 0\}$$

and consider the weight function $w' \colon E \to \mathbb{R}_0^+$ defined by

$$w'(e) \;=\; w(e) + C \qquad \text{for all } e \in E.$$

Now all bases of M have the same cardinality, say k. Let B be a basis; then the weights $w(B)$ and $w'(B)$ differ just by the constant kC. In particular, every basis of maximal weight for w' also has maximal weight for w. Hence we may use the greedy algorithm to find a basis B_0 of maximal weight for w' which is also a solution for the original problem (BMAX). Obviously, the greedy algorithm runs for w exactly as for w'; hence it yields the correct solution B_0 also when applied to the original function w. □

Theorem 5.5.2. *Let $M = (E, \mathbf{S})$ be a matroid, and let $w : E \to \mathbb{R}$ be any weight function on M. Then the greedy algorithm finds an optimal solution for the problem*

$$\text{(BMIN)} \qquad \textit{determine} \quad \min\{w(B) \colon B \textit{ is a basis of } M\},$$

provided that step (1) in Algorithm 5.1.1 is replaced as follows:

(1′) order the elements of E according to their weight:
$E = \{e_1, \ldots, e_m\}$ with $w(e_1) \le w(e_2) \le \ldots \le w(e_m)$.

Proof. This follows immediately from Theorem 5.5.1 by considering the weight function $-w$ instead of w. □

As an application, we investigate the behavior of the greedy algorithm in the context of duality. Suppose we are given a matroid $M = (E, \mathbf{S})$ and a weight function $w : E \to \mathbb{R}_0^+$. Obviously, a basis B of M has maximal weight if and only if the corresponding cobasis B^* of M^* has minimal weight. Now we use the greedy algorithm, modified as described in Theorem 5.5.2, to determine a basis B^* of M^* with minimal weight. Consider the moment when

we investigate the element e_k. Then e_k is added to the current solution – that is, the independent subset T^* constructed so far – if and only if $T^* \cup \{e_k\}$ is independent in M^*. Viewing this situation within M, we may as well say that e_k is removed from the current solution $T = E \setminus T^*$, as the (final) solution of the maximization problem for M is precisely the complement of the solution of the minimization problem for M^*. These considerations lead to the following *dual* version of the greedy algorithm, formulated in terms of the *primal* matroid M.

Algorithm 5.5.3 (dual greedy algorithm). Let $(E, \mathbf{S})$ be a matroid, and let $w\colon E \to \mathbb{R}_0^+$ be a weight function.

Procedure DUALGREEDY$(G, w; T)$

(1) order the elements of E according to their weight: $E = \{e_1, \ldots, e_m\}$ with $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$;
(2) $T \leftarrow E$;
(3) **for** $k = 1$ **to** m **do**
(4) **if** $(E \setminus T) \cup \{e_k\}$ does not contain a cocircuit
(5) **then** remove e_k from T
(6) **fi**
(7) **od**

Note that the condition in step (4) is satisfied if and only if $T^* \cup \{e_k\} = (E \setminus T) \cup \{e_k\}$ is independent in M^*; hence the correctness of the greedy algorithm immediately implies the following theorem.

Theorem 5.5.4. *Let $M = (E, \mathbf{S})$ be a matroid, and let $w : E \to \mathbb{R}_0^+$ be a nonnegative weight function on M. Then the dual greedy algorithm computes a basis B of $M = (E, \mathbf{S})$ with maximal weight.* □

Example 5.5.5. Let $M = M(G)$ be a graphic matroid, where G is connected. The dual greedy algorithm investigates the edges of G in the order given by increasing weight. Initially, $T = E$. When an edge e is examined, it is removed from the current solution T if and only if it does not form a cocycle with the edges already removed, that is, if removing e does not disconnect the graph (E, T). This special case of Algorithm 5.5.3 was already treated by Kruskal [Kru56].

In the remainder of this section, we look at the greedy algorithm as a possible approximation method for the problems described in Theorems 5.5.1 and 5.5.2 when $M = (E, \mathbf{S})$ is an arbitrary independence system, not necessarily a matroid. Unfortunately, this will not work well. Even for the maximization problem, the quotient of the weight of a solution found by the greedy algorithm and the weight of an optimal solution – which we used as a measure for the quality of approximation in Section 5.4 – does not make sense if negative weights occur. Still, there is one positive result: Theorem 5.4.2 carries over to the case of arbitrary weight functions if we consider the problem (P) of

Section 5.4, that is, if we require not a basis but only an independent set of maximal weight and terminate the greedy algorithm as soon as an element of negative weight would be chosen.

Let us now turn to the question whether there is a performance guarantee for applying the greedy algorithm to the minimization problem

(PMIN) determine $\min\{w(A)\colon A \text{ is a maximal independent set in } \mathbf{S}\}$,

where $w : E \to \mathbb{R}_0^+$ is a nonnegative weight function. Here the reciprocal quotient

$$g(M) = \min\left\{\frac{w(T_0)}{w(T_g)}\colon\ w\colon E \to \mathbb{R}_0^+\right\}$$

should be used for measuring the quality of approximation, where again T_g denotes a solution constructed by the greedy algorithm and T_0 an optimal solution for (PMIN). Clearly, the matroids are precisely the independence systems with $g(M) = 1$. Unfortunately, no result analogous to Theorem 5.4.2 can be proved; this was first shown in [KoHa78] via a rather trivial series of counterexamples, namely a path of length 2 with various weight functions. We will exhibit a class of considerably more interesting examples due to Reingold and Tarjan [ReTa81].

Example 5.5.6. Let us denote the complete graph on 2^t vertices by $G_t = (V_t, E_t)$. For each of these graphs, we define a weight function w_t satisfying the triangle inequality as follows. First we choose, for all $t \geq 2$, a Hamiltonian cycle C_t of G_t; for $t = 1$, we take C_1 as the only edge of G_1. We define w_t on C_t as indicated in Figure 5.2, where the edges not marked explicitly with their weight are understood to have weight 1.

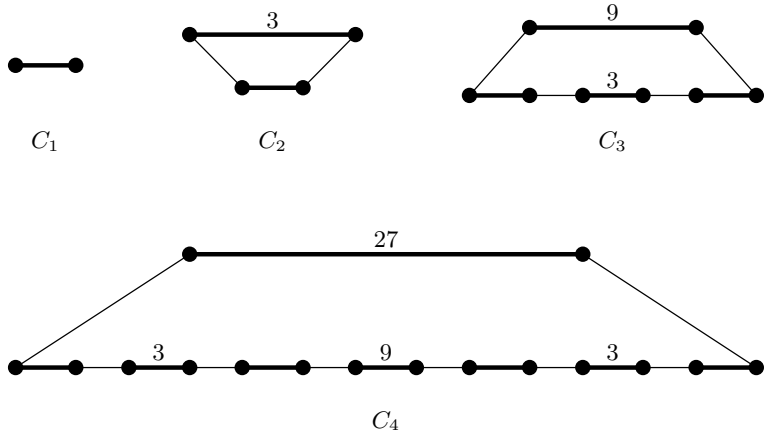


Fig. 5.2. A family of networks

For every edge $e = uv$ in $E_t \setminus C_t$, the weight $w_t(e)$ is defined as the distance $d_t(u, v)$ in the network (C_t, w_t). Since the largest weight occurring in C_t is

precisely the sum of the weights of all other edges of C_t (so that $w(C_t) = 2 \cdot 3^{t-1}$), it is easy to see that w_t indeed satisfies the triangle inequality.

Now put $M_t = (E_t, \mathbf{S}_t)$, where $\mathbf{S}_t$ is the set of all matchings of G_t. Thus we consider the problem of finding a maximal matching – that is, a 1-factor – of minimal weight for (G_t, w_t). It is easy to see that the greedy algorithm computes, for the cases $t = 1, 2, 3, 4$ shown in Figure 5.2, the 1-factors F_t drawn bold there, provided that we order edges of the same weight in a suitable manner; these 1-factors have weight 1, 4, 14, and 46, respectively. In the general case of an arbitrary t, one may show that a 1-factor F_t of G_t of weight

$$w_t(F_t) \;=\; 2 \times 3^{t-1} - 2^{t-1}$$

results; this formula for the weight of F_t can be derived from the rather obvious recursion

$$w_{t+1}(F_{t+1}) \;=\; 2w_t(F_t) - 3^{t-1} + 3^t,$$

where $w_1(F_1) = 1$. We leave the details to the reader. On the other hand, there is a 1-factor F_t' of G_t, for $t \geq 2$, of weight 2^{t-1} which consists of the edges not drawn bold in Figure 5.2: $F_t' = C_t \setminus F_t$. Thus the quality of the approximation found by the greedy algorithm is only

$$\frac{2^{t-1}}{2 \times 3^{t-1} - 2^{t-1}} \;\to\; 0 \qquad (\text{for } t \to \infty).$$

Example 5.5.6 shows that the greedy algorithm may yield an arbitrarily bad solution for (PMIN). By Theorem 5.4.4, the rank quotient of the independence system $(E_t, \mathbf{S}_t)$ formed by the matchings in G_t has value 2 for all $t \geq 2$. Hence in the case of minimization, the rank quotient does not guarantee a corresponding quality of approximation – a rather disappointing result.

It was shown in [ReTa81] that the bound for $w(T_0)/w(T_g)$ given in Example 5.5.6 is essentially worst possible. Indeed, for any weight function on the complete graph K_n satisfying the triangle inequality,

$$\frac{w(T_g)}{w(T_0)} \;\leq\; \left(\frac{3^\theta}{2^{\theta+1} - 1} - 1 \right) \times \frac{4}{3} n^{\log \frac{3}{2}},$$

where $\theta = \lceil \log n \rceil - \log n$. For the rather involved proof, we refer to the original paper.

Determining a 1-factor of minimal weight with respect to a weight function w on a complete graph satisfying the triangle inequality will be a tool for solving the *Chinese postman problem* in Chapter 14; this problem has interesting practical applications – for example, drawing large graphs with a plotter.

Exercise 5.5.7. Show that it is not possible to change the weight function w_t in Example 5.5.6 such that the quotient F_t'/F_t becomes smaller. Also, for an arbitrary weight function (not necessarily satisfying the triangle inequality), it is not possible to give any measure (as a function of n) for the quality of a 1-factor in a complete graph found by the greedy algorithm.

5.6 Accessible set systems

We conclude this chapter with a brief report on further generalizations of the greedy algorithm from independence systems to even more general systems of sets. As the methods used are rather similar to the methods we have been using (although more involved), we shall skip all proofs and refer the reader to the original literature instead.

A *set system* is simply a pair $M = (E, \mathbf{S})$, where E is a finite set and $\mathbf{S}$ is a nonempty subset of the power set of E. The elements of $\mathbf{S}$ are called *feasible sets* of M; maximal feasible sets will again be called *bases*. As the greedy algorithm always chooses single elements and adds them one by one to the feasible set under construction, it would not make sense to consider entirely arbitrary set systems. At the very least, we have to ensure that every feasible set can be obtained by successively adding single elements to the empty set. Formally, we require the following *accessibility axiom*:

(A) For each nonempty feasible set $X \in \mathbf{S}$, there exists an element $x \in X$ such that $X \setminus \{x\} \in \mathbf{S}$.

In particular, the empty set is contained in $\mathbf{S}$, as $\mathbf{S} \neq \emptyset$. A set system M satisfying axiom (A) is called an *accessible set system*. Any independence system is an accessible set system, but axiom (A) is a much weaker condition than the requirement of being closed under inclusion. Given an accessible set system M and a weight function $w : E \to \mathbb{R}$, we consider the following optimization problem:

(BMAX) determine $\max\{w(B) : B \text{ is a basis of } M\}$.

This generalizes the corresponding problem for independence systems. We also need to modify the greedy algorithm 5.1.1 so that it applies to accessible set systems.[8] This can be done as follows.

Algorithm 5.6.1. Let $M = (E, \mathbf{S})$ be an accessible set system, and let $w : E \to \mathbb{R}$ be a weight function.

Procedure GREEDY$(E, \mathbf{S}, w; T)$

(1) $T \leftarrow \emptyset$; $X \leftarrow E$;
(2) **while** there exists $x \in X$ with $T \cup \{x\} \in \mathbf{S}$ **do**
(3) choose some $x \in X$ with $T \cup \{x\} \in \mathbf{S}$ and
$w(x) \geq w(y)$ for all $y \in X$ with $T \cup \{y\} \in \mathbf{S}$;

[8] Note that it does not make sense to apply the original version of the greedy algorithm if $\mathbf{S}$ is not closed under inclusion: in this case, it might happen that an element x cannot be added to the feasible set T constructed so far, because $T \cup \{x\}$ is not feasible; nevertheless, it might be permissible to add x at some later point to the set $T' = T \cup A$. If $w(x) > w(y)$ for some $y \in A$, the original greedy algorithm 5.1.1 would fail in this situation, as the element x would already have been dismissed. To avoid this, we simply keep the strategy of always selecting the largest available element; all that is required is a different formulation.

(4) $T \leftarrow T \cup \{x\}$; $X \leftarrow X \setminus \{x\}$
(5) **od**

Of course, we want to characterize those accessible set systems for which Algorithm 5.6.1 always finds an optimal solution for (BMAX). Before describing this result, we consider a special class of accessible set systems introduced by Korte and Lovász [KoLo81].

An accessible set system M satisfying the exchange axiom (2) of Theorem 5.2.1 is called a *greedoid*. Greedoids have been studied intensively because many interesting objects in combinatorics and optimization are greedoids. In particular, the so-called *antimatroids* are greedoids. Antimatroids constitute a combinatorial abstraction of the notion of convexity; they play an important role in convexity, partially ordered sets, and graph theory. Greedoids occur as well in the context of matchings and of Gauß elimination. We will not go into detail here, but recommend that the reader consult the extensive survey [BjZi92] or the monograph [KoLS91]. Unfortunately, the greedy algorithm does not find an optimal solution of (BMAX) for all greedoids.[9] However, Korte and Lovász were able to characterize those greedoids for which the greedy algorithm works. There is a simpler characterization due to Bryant and Brooksbank [BrBr92], which uses the following *strong exchange axiom*. We note that this condition holds for every matroid, but not for all greedoids.

(SE) For $J, K \in \mathbf{S}$ with $|J| = |K| + 1$, there always exists some $a \in J \setminus K$ such that $K \cup \{a\}$ and $J \setminus \{a\}$ are in $\mathbf{S}$.

Result 5.6.2. *Let $M = (E, \mathbf{S})$ be a greedoid. Then the greedy algorithm 5.6.1 finds an optimal solution of (BMAX) for all weight functions $w : E \to \mathbb{R}$ if and only if M satisfies axiom (SE).* □

We need some further preparations to be able to formulate the characterization of those accessible set systems $M = (E, \mathbf{S})$ for which the greedy algorithm computes an optimal solution. Given a feasible set A, we write

$$\mathrm{ext}(A) := \{x \in E \setminus A : A \cup \{x\} \in \mathbf{S}\}.$$

Now there are some situations where the greedy algorithm does not even construct a basis, but stops with some feasible set which is not maximal. This happens if there exists a basis B with a proper feasible subset $A \subset B$ such that $\mathrm{ext}(A) = \emptyset$. In this case, we may define a weight function w by

$$w(x) := \begin{cases} 2 & \text{for } x \in A, \\ 1 & \text{for } x \in B \setminus A, \\ 0 & \text{otherwise;} \end{cases}$$

[9] Characterizing greedoids in terms of the greedy algorithm requires the use of certain non-linear objective functions; see [KoLS91].

then the greedy algorithm constructs A, but cannot extend A to the optimal basis B. The accessibility axiom (A) is too weak to prevent such situations: it merely ensures that a basis B can be obtained somehow by adding single elements successively to the empty set, but not necessarily by adding elements to a *given* feasible subset of B. To avoid this, we require the following *extensibility axiom*:

(E) For every basis B and every feasible subset $A \subset B$ with $A \neq B$, there exists some $x \in B \setminus A$ with $A \cup \{x\} \in \mathbf{S}$.

Note that this axiom is satisfied for all greedoids. We need one more definition. For any set system $M = (E, \mathbf{S})$, define

$$\overline{\mathbf{S}} := \{X \subseteq E : \text{ there is } A \in \mathbf{S} \text{ with } X \subseteq A\},$$

and call $\overline{M} := (E, \overline{\mathbf{S}})$ the *hereditary closure* of M. Now we require the following *closure congruence axiom*:

(CC) For every feasible set A, for all $x, y \in \text{ext}\ (A)$, and for each subset $X \subseteq E \setminus (A \cup \text{ext}(A))$, $A \cup X \cup \{x\} \in \overline{\mathbf{S}}$ implies $A \cup X \cup \{y\} \in \overline{\mathbf{S}}$.

Exercise 5.6.3. Show that an accessible set system $M = (E, \mathbf{S})$ for which the greedy algorithm works correctly has to satisfy axiom (CC).

One may show that axiom (CC) is independent of the exchange axiom, even if we only consider accessible set systems satisfying the extensibility axiom. In fact, there are greedoids not satisfying (CC); on the other hand, independence systems always satisfy (CC), because the only choice for X is $X = \emptyset$. We need one final axiom:

(ME) The hereditary closure $\overline{M}$ of M is a matroid.

(ME) is called the *matroid embedding axiom*. Now we can state the following characterization due to Helman, Mont and Shapiro [HeMS93]:

Result 5.6.4. *Let $M = (E, \mathbf{S})$ be an accessible set system. Then the following statements are equivalent:*

(1) *M satisfies axioms (E), (CC) and (ME).*
(2) *For every weight function $w : E \to \mathbb{R}$, the optimal solutions of (BMAX) are precisely those bases of M which are found by the greedy algorithm 5.6.1 (given an appropriate order of the elements of equal weight).*
(3) *For every weight function $w : E \to \mathbb{R}$, the greedy algorithm 5.6.1 yields an optimal solution of (BMAX).* □

The reader might try to fill in the missing parts of the proof; this is a more demanding exercise, but can be done using the methods we have presented. Alternatively, we recommend a look at the original paper [HeMS93], which contains some further interesting results. In particular, the authors consider *bottleneck problems*, that is, problems of the form

(BNP) Maximize $\min\{w(x)\colon x \in B\}$ over all bases B of M

for a given weight function $w : E \to \mathbb{R}$. The greedy algorithm constructs an optimal solution for (BNP) in the situation of Result 5.6.4. In fact, this holds even under considerably weaker conditions. We need one further axiom, namely the *strong extensibility axiom*:

(SE) For every basis B and each feasible set A with $|A| < |B|$, there exists $x \in B \setminus A$ with $A \cup \{x\} \in \mathbf{S}$.

Then the following characterization holds [HeMS93]:

Result 5.6.5. *Let $M = (E, \mathbf{S})$ be an accessible set system. The greedy algorithm 5.6.1 constructs an optimal solution for (BNP) for all weight functions $w\colon E \to \mathbb{R}$ if and only if M satisfies axiom (SE).* □

For partially ordered set systems, the greedy algorithm was studied by Faigle [Fai79] who obtained characterizations analogous to Results 5.6.4 and 5.6.5. Further characterizations of related structures by the greedy algorithm (or appropriately modified versions) can be found in [Fai85], [Goe88], and [BoFa90], where ordered languages, greedoids of Gauß elimination, and antimatroids are studied, respectively. There are further important generalization of the notion of a matroid such as *oriented matroids*. We will not consider these structures here, but refer the reader to the monographs [BaKe92] and [BjLSW92].

6

Flows

What need you flow so fast?
ANONYMOUS

In this chapter, we investigate *flows* in networks: How much can be transported in a network from a *source* s to a *sink* t if the *capacities* of the connections are given? Such a network might model a system of pipelines, a water supply system, or a system of roads. With its many applications, the theory of flows is one of the most important parts of combinatorial optimization. In Chapter 7 we will encounter several applications of the theory of flows within combinatorics, and flows and related notions will appear again and again throughout the book. The once standard reference, *Flows in Networks* by Ford and Fulkerson [FoFu62], is still worth reading; an extensive, more recent treatment is provided in [AhMO93].

6.1 The theorems of Ford and Fulkerson

In this chapter, we study networks of the following special kind. Let $G = (V, E)$ be a digraph, and let $c: E \to \mathbb{R}_0^+$ be a mapping; the value $c(e)$ will be called the *capacity* of the edge e. Moreover, let s and t be two special vertices of G such that t is accessible from s.[1] Then $N = (G, c, s, t)$ is called a *flow network* with *source* s and *sink* t. An *admissible flow* or, for short, a *flow* on N is a mapping $f: E \to \mathbb{R}_0^+$ satisfying the following two conditions:

(F1) $0 \leq f(e) \leq c(e)$ for each edge e;

(F2) $\sum_{e^+=v} f(e) = \sum_{e^-=v} f(e)$ for each vertex $v \neq s, t$, where e^- and e^+ denote the start and end vertex of e, respectively.

Thus the *feasibility condition* (F1) requires that each edge carries a nonnegative amount of flow which may not exceed the capacity of the edge, and

[1] Some authors require in addition $d_{\text{in}}(s) = d_{\text{out}}(t) = 0$. We do not need this condition here; it would also be inconvenient for our investigation of symmetric networks and the network synthesis problem in Chapter 12.

the *flow conservation condition* (F2) means that flows are preserved: at each vertex, except for the source and the sink, the amount that flows in also flows out. It is intuitively clear that the total flow coming out of s should be the same as the total flow going into t; let us provide a formal proof.

Lemma 6.1.1. *Let $N = (G, c, s, t)$ be a flow network with flow f. Then*

$$\sum_{e^-=s} f(e) - \sum_{e^+=s} f(e) \;=\; \sum_{e^+=t} f(e) - \sum_{e^-=t} f(e). \tag{6.1}$$

Proof. Trivially,

$$\sum_{e^-=s} f(e) + \sum_{e^-=t} f(e) + \sum_{v\neq s,t} \sum_{e^-=v} f(e) \;=\; \sum_{e} f(e) \;=$$
$$= \sum_{e^+=s} f(e) + \sum_{e^+=t} f(e) + \sum_{v\neq s,t} \sum_{e^+=v} f(e).$$

Now the assertion follows immediately from (F2). □

The quantity in equation (6.1) is called the *value* of f; it is denoted by $w(f)$. A flow f is said to be *maximal* if $w(f) \geq w(f')$ holds for every flow f' on N. The main problem studied in the theory of flows is the determination of a maximal flow in a given network. Note that, a priori, it is not entirely obvious that maximal flows always exist; however, we will soon see that this is indeed the case.

Let us first establish an upper bound for the value of an arbitrary flow. We need some definitions. Let $N = (G, c, s, t)$ be a flow network. A *cut* of N is a partition $V = S \mathbin{\dot\cup} T$ of the vertex set V of G into two disjoint sets S and T with $s \in S$ and $t \in T$; thus cuts in flow networks constitute a special case of the cuts of $|G|$ introduced in Section 4.3. The *capacity* of a cut (S, T) is defined as

$$c(S, T) \;=\; \sum_{e^-\in S, e^+\in T} c(e);$$

thus it is just the sum of the capacities of all those edges e in the corresponding cocycle $E(S, T)$ which are oriented from S to T. The cut (S, T) is said to be *minimal* if $c(S, T) \leq c(S', T')$ holds for every cut (S', T'). An edge e is called *saturated* if $f(e) = c(e)$, and *void* if $f(e) = 0$.

The following lemma shows that the capacity of a minimal cut gives the desired upper bound on the value of a flow; moreover, we can also characterize the case of equality in this bound.

Lemma 6.1.2. *Let $N = (G, c, s, t)$ be a flow network, (S, T) a cut, and f a flow. Then*

$$w(f) \;=\; \sum_{e^-\in S, e^+\in T} f(e) - \sum_{e^+\in S, e^-\in T} f(e). \tag{6.2}$$

In particular, $w(f) \leq c(S, T)$; equality holds if and only if each edge e with $e^- \in S$ and $e^+ \in T$ is saturated, whereas each edge e with $e^- \in T$ and $e^+ \in S$ is void.

Proof. Summing equation (F2) over all $v \in S$ gives

$$\begin{aligned} w(f) &= \sum_{v \in S} \left(\sum_{e^- = v} f(e) - \sum_{e^+ = v} f(e) \right) \\ &= \sum_{e^- \in S, e^+ \in S} f(e) - \sum_{e^+ \in S, e^- \in S} f(e) + \sum_{e^- \in S, e^+ \in T} f(e) - \sum_{e^+ \in S, e^- \in T} f(e). \end{aligned}$$

The first two terms add to 0. Now note $f(e) \leq c(e)$ for all edges e with $e^- \in S$ and $e^+ \in T$, and $f(e) \geq 0$ for all edges e with $e^+ \in S$ and $e^- \in T$. This implies the desired inequality and also the assertion on the case of equality. □

The main result of this section states that the maximal value of a flow always equals the minimal capacity of a cut. But first we characterize the maximal flows. We need a further definition. Let f be a flow in the network $N = (G, c, s, t)$. A path W from s to t is called an *augmenting path* with respect to f if $f(e) < c(e)$ holds for every forward edge $e \in W$, whereas $f(e) > 0$ for every backward edge $e \in W$. The following three fundamental theorems are due to Ford and Fulkerson [FoFu56].

Theorem 6.1.3 (augmenting path theorem). *A flow f on a flow network $N = (G, c, s, t)$ is maximal if and only if there are no augmenting paths with respect to f.*

Proof. First let f be a maximal flow. Suppose there is an augmenting path W. Let d be the minimum of all values $c(e) - f(e)$ (taken over all forward edges e in W) and all values $f(e)$ (taken over the backward edges in W). Then $d > 0$, by definition of an augmenting path. Now we define a mapping $f' \colon E \to \mathbb{R}_0^+$ as follows:

$$f'(e) = \begin{cases} f(e) + d & \text{if } e \text{ is a forward edge in } W, \\ f(e) - d & \text{if } e \text{ is a backward edge in } W, \\ f(e) & \text{otherwise.} \end{cases}$$

It is easily checked that f' is a flow on N with value $w(f') = w(f) + d > w(f)$, contradicting the maximality of f.

Conversely, suppose there are no augmenting paths in N with respect to f. Let S be the set of all vertices v such that there exists an augmenting path from s to v (including s itself), and put $T = V \setminus S$. By hypothesis, (S, T) is a cut of N. Note that each edge $e = uv$ with $e^- = u \in S$ and $e^+ = v \in T$ has to be saturated: otherwise, it could be appended to an augmenting path from s to u to reach the point $v \in T$, a contradiction. Similarly, each edge e with $e^- \in T$ and $e^+ \in S$ has to be void. Then Lemma 6.1.2 gives $w(f) = c(S, T)$, so that f is maximal. □

Let us note the following useful characterization of maximal flows contained in the preceding proof:

Corollary 6.1.4. *Let f be a flow on a flow network $N = (G, c, s, t)$, denote by S_f the set of all vertices accessible from s on an augmenting path with respect to f, and put $T_f = V \setminus S_f$. Then f is a maximal flow if and only if $t \in T_f$. In this case, (S_f, T_f) is a minimal cut: $w(f) = c(S_f, T_f)$.* □

Theorem 6.1.5 (integral flow theorem). *Let $N = (G, c, s, t)$ be a flow network where all capacities $c(e)$ are integers. Then there is a maximal flow on N such that all values $f(e)$ are integral.*

Proof. By setting $f_0(e) = 0$ for all e, we obtain an integral flow f_0 on N with value 0. If this trivial flow is not maximal, then there exists an augmenting path with respect to f_0. In that case the number d appearing in the proof of Theorem 6.1.3 is a positive integer, and we can construct an integral flow f_1 of value d as in the proof of Theorem 6.1.3. We continue in the same manner. As the value of the flow is increased in each step by a positive integer and as the capacity of any cut is an upper bound on the value of the flow (by Lemma 6.1.2), after a finite number of steps we reach an integral flow f for which no augmenting path exists. By Theorem 6.1.3, this flow f is maximal. □

Theorem 6.1.6 (max-flow min-cut theorem). *The maximal value of a flow on a flow network N is equal to the minimal capacity of a cut for N.*

Proof. If all capacities are integers, the assertion follows from Theorem 6.1.5 and Corollary 6.1.4. The case where all capacities are rational can be reduced to the integral case by multiplying all numbers by their common denominator. Then real-valued capacities may be treated using a continuity argument, since the set of flows is a compact subset of $\mathbb{R}^{|E|}$ and since $w(f)$ is a continuous function of f. A different, constructive proof for the real case is provided by the theorem of Edmonds and Karp [EdKa72], which we will treat in the next section. □

Theorem 6.1.6 was obtained in [FoFu56] and, independently, in [ElFS56]. In practice, real capacities do not occur, as a computer can only represent (a finite number of) rational numbers anyway. From now on, we mostly restrict ourselves to integral flows. Sometimes we also allow networks on directed multigraphs; this is not really more general, because parallel edges can be replaced by a single edge whose capacity is the sum of the corresponding capacities of the parallel edges.

The remainder of this chapter deals with several algorithms for finding a maximal flow. The proof of Theorem 6.1.5 suggests the following rough outline of such an algorithm:

(1) $f(e) \leftarrow 0$ for all edges e;
(2) **while** there exists an augmenting path with respect to f **do**
(3) let $W = (e_1, \ldots, e_n)$ be an augmenting path from s to t;
(4) $d \leftarrow \min\big(\{c(e_i) - f(e_i) : e_i \text{ is a forward edge in } W\}$
$\cup\ \{f(e_i) : e_i \text{ is a backward edge in } W\}\big)$;

(5) $f(e_i) \leftarrow f(e_i) + d$ for each forward edge e_i;
(6) $f(e_i) \leftarrow f(e_i) - d$ for each backward edge e_i;
(7) **od**

Of course, we still have to specify a technique for finding augmenting paths. We will apply a modified breadth first search (BFS), where edges may be used regardless of their orientation as long as they satisfy the necessary condition $f(e) < c(e)$ or $f(e) > 0$. Note that we not only have to decide whether t is accessible from s by an augmenting path, but we also need to find the value for d and change the values $f(e)$ accordingly. In view of these additional requirements, it makes sense to use a labelling technique; moreover, this will also allow us to find a minimal cut easily.

Algorithm 6.1.7 (labelling algorithm of Ford and Fulkerson). Let $N = (G, c, s, t)$ be a flow network.

Procedure FORDFULK($N; f, S, T$)

(1) **for** $e \in E$ **do** $f(e) \leftarrow 0$ **od**
(2) label s with $(-, \infty)$;
(3) **for** $v \in V$ **do** $u(v) \leftarrow$ false; $d(v) \leftarrow \infty$ **od**
(4) **repeat**
(5) choose a vertex v which is labelled and satisfies $u(v) =$ false;
(6) **for** $e \in \{e \in E : e^- = v\}$ **do**
(7) **if** $w = e^+$ is not labelled **and** $f(e) < c(e)$ **then**
(8) $d(w) \leftarrow \min\{c(e) - f(e), d(v)\}$; label w with $(v, +, d(w))$ **fi**
(9) **od**
(10) **for** $e \in \{e \in E : e^+ = v\}$ **do**
(11) **if** $w = e^-$ is not labelled **and** $f(e) > 0$ **then**
(12) $d(w) \leftarrow \min\{f(e), d(v)\}$; label w with $(v, -, d(w))$ **fi**
(13) **od**
(14) $u(v) \leftarrow$ true;
(15) **if** t is labelled
(16) **then** let d be the last component of the label of t;
(17) $w \leftarrow t$;
(18) **while** $w \neq s$ **do**
(19) find the first component v of the label of w;
(20) **if** the second component of the label of w is $+$
(21) **then** set $f(e) \leftarrow f(e) + d$ for $e = vw$
(22) **else** set $f(e) \leftarrow f(e) - d$ for $e = wv$
(23) **fi**
(24) $w \leftarrow v$
(25) **od**
(26) delete all labels except for the label of s;
(27) **for** $v \in V$ **do** $d(v) \leftarrow \infty$; $u(v) \leftarrow$ false **od**
(28) **fi**

(29) **until** $u(v) =$ true for all vertices v which are labelled;
(30) let S be the set of vertices which are labelled and put $T \leftarrow V \setminus S$

Using the proofs we gave for Theorems 6.1.3 and 6.1.5, we immediately get the following theorem due to Ford and Fulkerson [FoFu57].

Theorem 6.1.8. *Let N be a network whose capacity function c takes only integral (or rational) values. Then Algorithm 6.1.7 determines a maximal flow f and a minimal cut (S,T), so that $w(f) = c(S,T)$ holds.* □

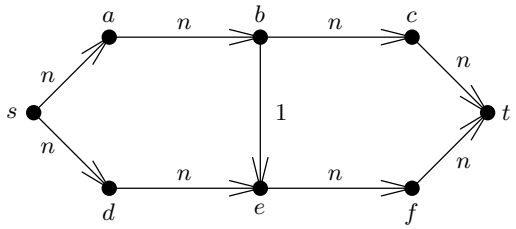


Fig. 6.1. A flow network

Algorithm 6.1.7 may fail for irrational capacities, if the vertex v in step (5) is chosen in an unfortunate way; an example for this can be found in [FoFu62], p. 21. In that example the algorithm not only does not terminate, but it even converges to a value which is only 1/4 of the maximal possible flow value. Moreover, Algorithm 6.1.7 is not polynomial even for integer capacities, because the number of necessary changes of the flow f does not only depend on $|V|$ and $|E|$, but might also depend on c. For example, if we use the paths

$$s \text{ — } a \text{ — } b \text{ — } e \text{ — } f \text{ — } t \qquad \text{and} \qquad s \text{ — } d \text{ — } e \text{ — } b \text{ — } c \text{ — } t$$

alternately as augmenting paths for the network in Figure 6.1 (which the algorithm will do if vertex v in step (5) is chosen suitably), the value of the flow will only be increased by 1 in each step, so that we need $2n$ iterations. Of course, this can be avoided by choosing the paths appropriately; with

$$s \text{ — } a \text{ — } b \text{ — } c \text{ — } t \qquad \text{and} \qquad s \text{ — } d \text{ — } e \text{ — } f \text{ — } t,$$

we need only two iterations. In the next section, we show how the augmenting paths can be chosen efficiently. Then we shall also apply the resulting algorithm to an example and show the computations in detail. We close this section with some exercises.

Exercise 6.1.9. Let $N = (G, c, s, t)$ be a flow network for which the *capacities* of the vertices are likewise restricted: there is a further mapping $d\colon V \to \mathbb{R}_0^+$, and the flows f have to satisfy the additional restriction

(F3) $$\sum_{e^+=v} f(e) \;\leq\; d(v) \quad \text{for } v \neq s,t.$$

For instance, we might consider an irrigation network where the vertices are pumping stations with limited capacity. Reduce this problem to a problem for an appropriate ordinary flow network and generalize Theorem 6.1.6 to this situation; see [FoFu62], §1.11.

Exercise 6.1.10. How can the case of several sources and several sinks be treated?

Exercise 6.1.11. Let $N = (G, c, s, t)$ be a flow network, and assume that N admits flows of value $\neq 0$. Show that there exists at least one edge e in N whose removal decreases the value of a maximal flow on N. An edge e is called *most vital* if the removal of e decreases the maximal flow value as much as possible. Is an edge of maximal capacity in a minimal cut necessarily most vital?

Exercise 6.1.12. By Theorem 6.1.5, a flow network with integer capacities always admits an integral maximal flow. Is it true that *every* maximal flow has to be integral?

Exercise 6.1.13. Let f be a flow in a flow network N. The *support* of f is $\operatorname{supp} f = \{e \in E \colon f(e) \neq 0\}$. A flow f is called *elementary* if its support is a path. The proof of Theorem 6.1.6 and the algorithm of Ford and Fulkerson show that there exists a maximal flow which is the sum of elementary flows. Can every maximal flow be represented by such a sum?

Exercise 6.1.14. Modify the process of labelling the vertices in Algorithm 6.1.7 in such a way that the augmenting path chosen always has maximal possible capacity (so that the value of the flow is always increased as much as possible).

Hint: Use an appropriate variation of the algorithm of Dijkstra.

Exercise 6.1.15. Let $N = (G, c, s, t)$ be a flow network, and assume that both (S, T) and (S', T') are minimal cuts for N. Show that $(S \cap S', T \cup T')$ and $(S \cup S', T \cap T')$ are likewise minimal cuts for N.

Exercise 6.1.16. Let f be any maximal flow. Shot that the set S_f defined in Corollary 6.1.4 is the intersection over all sets S for which $(S, V \setminus S)$ is a minimal cut. In particular, S_f does not depend on the choice of the maximal flow f.

6.2 The algorithm of Edmonds and Karp

As we have seen in the previous section, the labelling algorithm of Ford and Fulkerson is, in general, not polynomial. We now consider a modification of this algorithm due to Edmonds and Karp [EdKa72] for which we can prove a polynomial complexity, namely $O(|V||E|^2)$. As we will see, it suffices if we

always use an augmenting path of shortest length – that is, having as few edges as possible – for increasing the flow. To find such a path, we just make step (5) in Algorithm 6.1.7 more specific: we require that the vertex v with $u(v) =$ false which was labelled first is chosen. Then the labelling process proceeds as for a BFS; compare Algorithm 3.3.1. This principle for selecting the vertex v is also easy to implement: we simply collect all labelled vertices in a queue – that is, some vertex w is appended to the queue when it is labelled in step (8) or (12). This simple modification is enough to prove the following result.

Theorem 6.2.1. *Replace step (5) in Algorithm 6.1.7 as follows:*

(5′) among all vertices with $u(v) =$ false, let v be the vertex which was labelled first.

Then the resulting algorithm has complexity $O(|V||E|^2)$.

Proof. We have already noted that the flow f is always increased using an augmenting path of shortest length, provided that we replace step (5) by (5′). Let f_0 be the flow of value 0 defined in step (1), and let $f_1, f_2, f_3, \ldots$ be the sequence of flows constructed subsequently. Denote the shortest length of an augmenting path from s to v with respect to f_k by $x_v(k)$. We begin by proving the inequality

$$x_v(k+1) \geq x_v(k) \quad \text{for all } k \text{ and } v. \tag{6.3}$$

By way of contradiction, suppose that (6.3) is violated for some pair (v, k); we may assume that $x_v(k+1)$ is minimal among the $x_w(k+1)$ for which (6.3) does not hold. Consider the last edge e on a shortest augmenting path from s to v with respect to f_{k+1}. Suppose first that e is a forward edge, so that $e = uv$ for some vertex u; note that this requires $f_{k+1}(e) < c(e)$. Now $x_v(k+1) = x_u(k+1) + 1$, so that $x_u(k+1) \geq x_u(k)$ by our choice of v. Hence $x_v(k+1) \geq x_u(k) + 1$. On the other hand, $f_k(e) = c(e)$, as otherwise $x_v(k) \leq x_u(k) + 1$ and $x_v(k+1) \geq x_v(k)$, contradicting the choice of v. Therefore e was as a backward edge when f_k was changed to f_{k+1}. As we have used an augmenting path of shortest length for this change, we conclude $x_u(k) = x_v(k) + 1$ and hence $x_v(k+1) \geq x_v(k) + 2$, a contradiction. The case where e is a backward edge can be treated in the same manner. Moreover, similar arguments also yield the inequality

$$y_v(k+1) \geq y_v(k) \quad \text{for all } k \text{ and } v, \tag{6.4}$$

where $y_v(k)$ denotes the length of a shortest augmenting path from v to t with respect to f_k.

When increasing the flow, the augmenting path always contains at least one *critical* edge: the flow through this edge is either increased up to its capacity or decreased to 0. Let $e = uv$ be a critical edge in the augmenting path with respect to f_k; this path consists of $x_v(k) + y_v(k) = x_u(k) + y_u(k)$ edges. If e is used the next time in some augmenting path (with respect to

f_h, say), it has to be used in the opposite direction: if e was a forward edge for f_k, it has to be a backward edge for f_h, and vice versa.

Suppose that e was a forward edge for f_k. Then $x_v(k) = x_u(k) + 1$ and $x_u(h) = x_v(h) + 1$. By (6.3) and (6.4), $x_v(h) \geq x_v(k)$ and $y_u(h) \geq y_u(k)$. Hence we obtain

$$x_u(h) + y_u(h) = x_v(h) + 1 + y_u(h) \geq x_v(k) + 1 + y_u(k) = x_u(k) + y_u(k) + 2.$$

Thus the augmenting path with respect to f_h is at least two edges longer than the augmenting path with respect to f_k. This also holds for the case where e was a backward edge for f_h; to see this, exchange the roles of u and v in the preceding argument. Trivially, no augmenting path can contain more than $|V| - 1$ edges. Hence each edge can be critical at most $(|V| - 1)/2$ times, and thus the flow can be changed at most $O(|V||E|)$ times. (In particular, this establishes that the algorithm has to terminate even if the capacities are non-rational.) Each iteration – finding an augmenting path and updating the flow – takes only $O(|E|)$ steps, since each edge is treated at most three times: twice during the labelling process and once when the flow is changed. This implies the desired complexity bound of $O(|V||E|^2)$. □

Remark 6.2.2. As the cardinality of E is between $O(|V|)$ and $O(|V|^2)$, the complexity of the algorithm of Edmonds and Karp lies between $O(|V|^3)$ for sparse graphs (hence, in particular, for planar graphs) and $O(|V|^5)$ for dense graphs.

Examples for networks with n vertices and $O(n^2)$ edges for which the algorithm of Edmonds and Karp actually needs $O(n^3)$ iterations may be found in [Zad72, Zad73b]; thus the estimates used in the proof of Theorem 6.2.1 are best possible. Of course, this by no means precludes the existence of more efficient algorithms. One possible approach is to look for algorithms which are not based on the use of augmenting paths; we will see examples for this approach in Sections 6.4 and 6.6 as well as in Chapter 11. Another idea is to combine the iterations in a clever way into larger phases; for instance, it turns out to be useful to consider all augmenting paths of a constant length in one block; see Sections 6.3 and 6.4. Not surprisingly, such techniques are not only better but also more involved.

Example 6.2.3. We use the algorithm of Edmonds and Karp to determine a maximal flow and a minimal cut in the network N given in Figure 6.2. The capacities are given there in parentheses; the numbers without parentheses in the following figures always give the respective values of the flow. We also state the labels which are assigned at the respective stage of the algorithm; when examining the possible labellings coming from some vertex v on forward edges (steps (6) through (9)) and on backward edges (steps (10) through (13)), we consider the adjacent vertices in alphabetical order, so that the course of the algorithm is uniquely determined. The augmenting path used for the construction of the next flow is drawn bold.

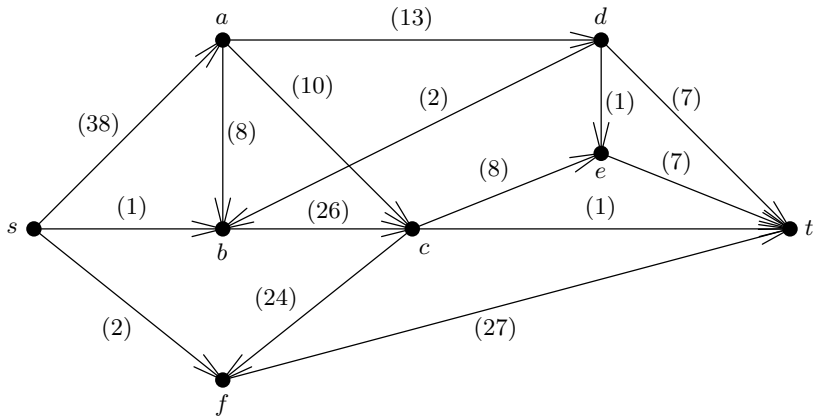


Fig. 6.2. A flow network

We start with the zero flow f_0, that is, $w(f_0) = 0$. The vertices are labelled in the order a, b, f, c, d, t as shown in Figure 6.3; e is not labelled because t is reached before e is considered. Figures 6.3 to 6.12 show how the algorithm proceeds. Note that the final augmenting path uses a backward edge, see Figure 6.11. In Figure 6.12, we have also indicated the minimal cut (S_f, T_f) associated with the maximal flow $f = f_9$ according to Corollary 6.1.4.

The alert reader will note that many labels are not changed from one iteration to the next. As all the labels are deleted in step (26) of the algorithm after each change of the flow, this means we do a lot of unnecessary calculations. It is possible to obtain algorithms of better complexity by combining the changes of the flow into bigger *phases*. To do this, a *blocking flow* is constructed in some appropriate auxiliary network. This important approach will be treated in Sections 6.3 and 6.4.

Exercise 6.2.4. Show that the flow network $N = (G, c, s, t)$ in Figure 6.2 admits two distinct integral maximal flows. Decide whether or not N contains two distinct minimal cuts. Hint: Recall Exercise 6.1.16.

We mention that Edmonds and Karp have also shown that the flow has to be changed at most $O(\log w)$ times, where w is the maximal value of a flow on N, if we always choose an augmenting path of maximal capacity. Even though we do not know w a priori, the number of steps necessary for this method is easy to estimate, as w is obviously bounded by

$$W = \min \left\{ \sum_{e^-=s} c(e), \sum_{e^+=t} c(e) \right\}.$$

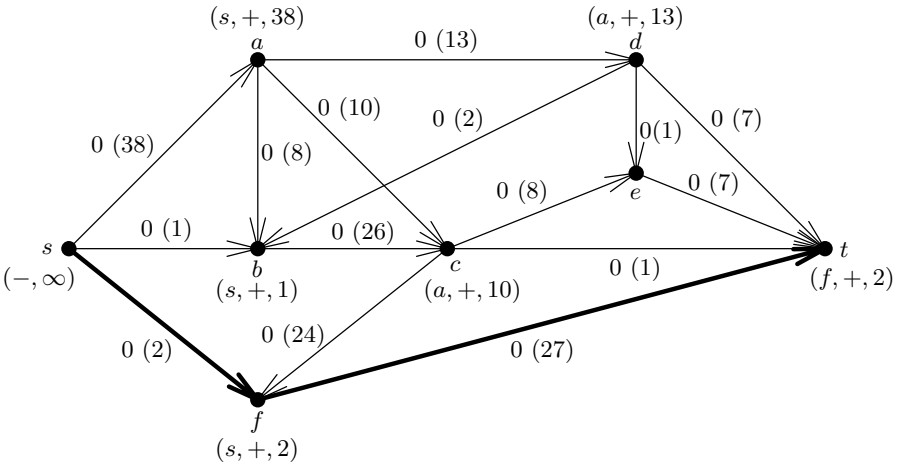


Fig. 6.3. $w(f_0) = 0$

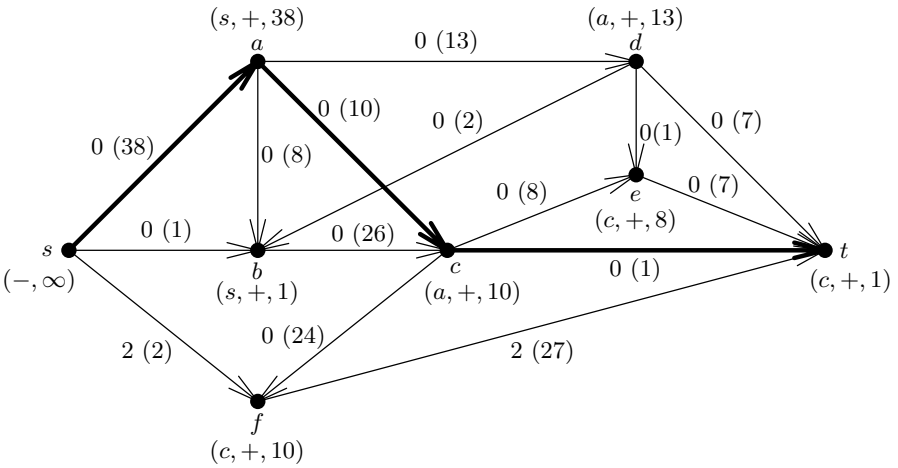


Fig. 6.4. $w(f_1) = 2$

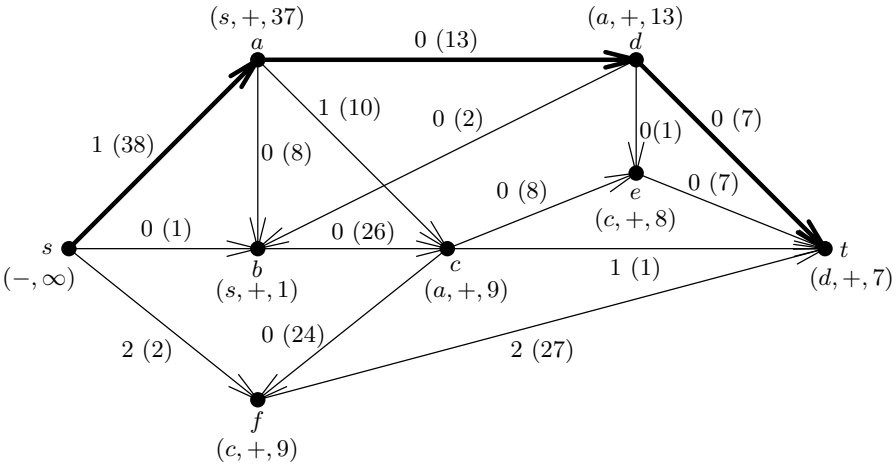


Fig. 6.5. $w(f_2) = 3$

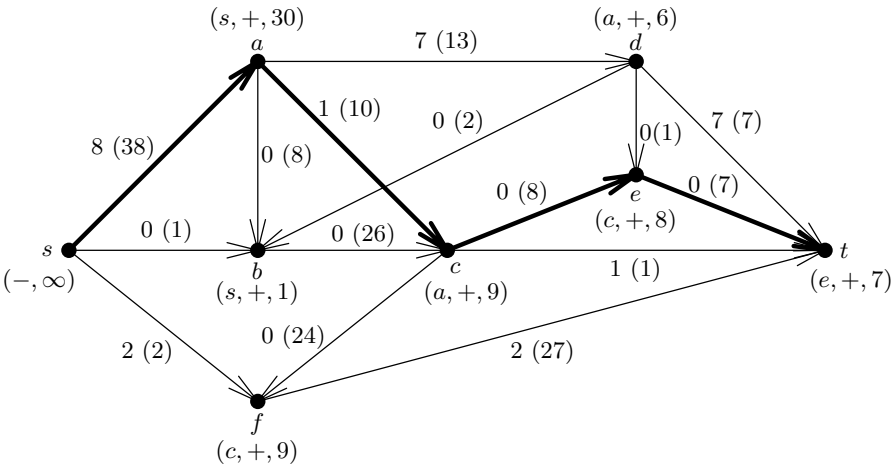


Fig. 6.6. $w(f_3) = 10$

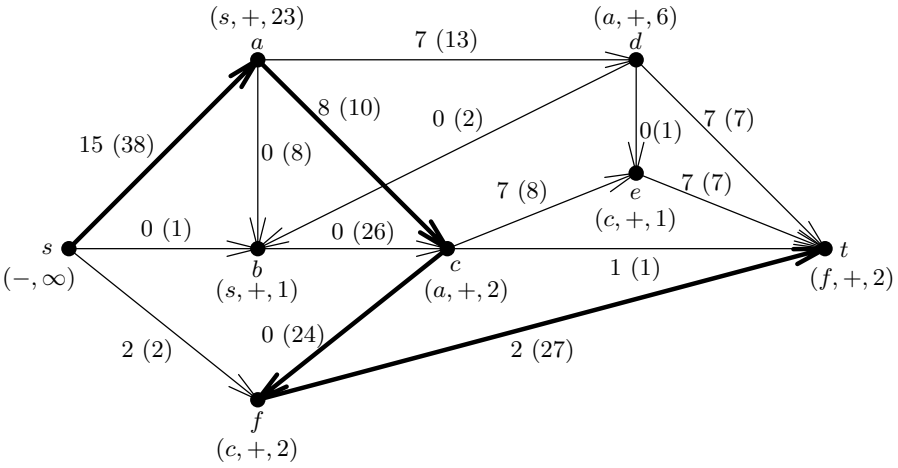


Fig. 6.7. $w(f_4) = 17$

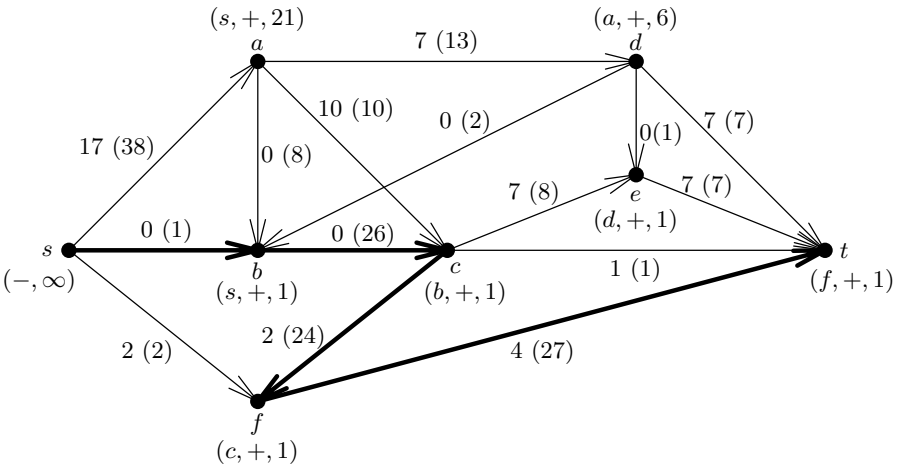


Fig. 6.8. $w(f_5) = 19$

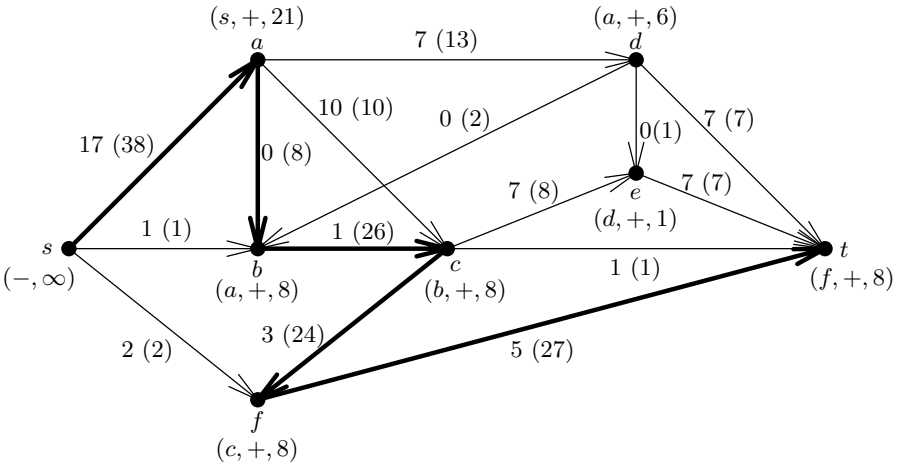


Fig. 6.9. $w(f_6) = 20$

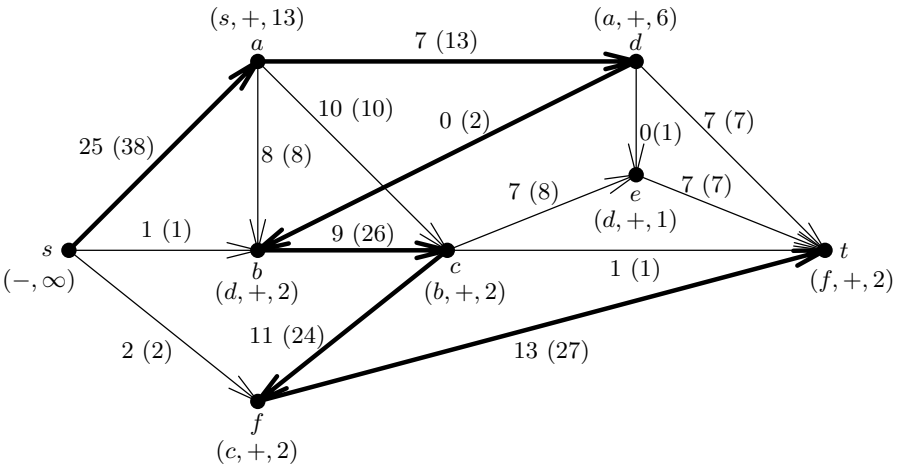


Fig. 6.10. $w(f_7) = 28$

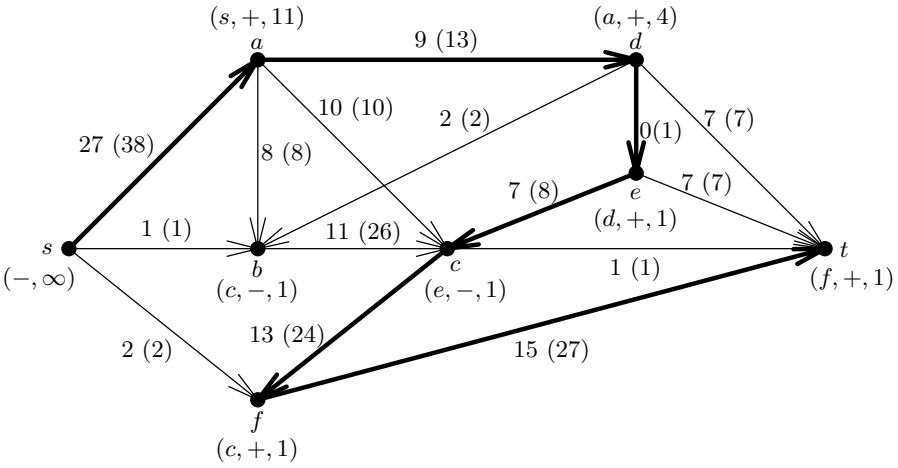


Fig. 6.11. $w(f_8) = 30$

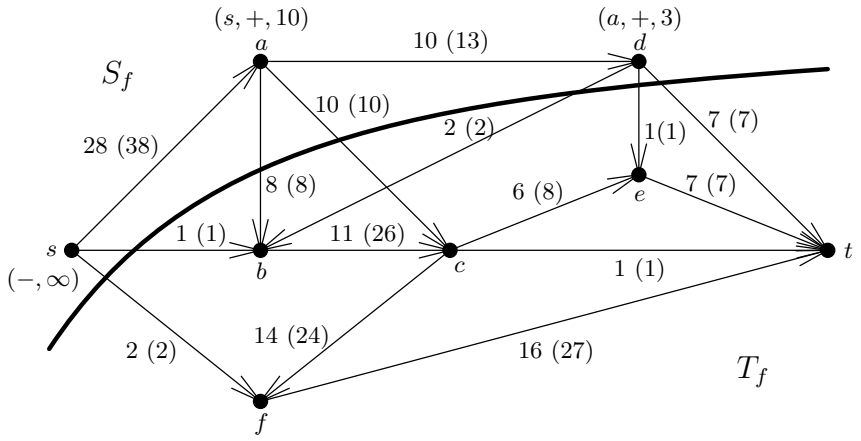


Fig. 6.12. $w(f_9) = 31 = c(S_f, T_f)$

Note that this approach does not yield a polynomial algorithm, since the bound depends also on the capacities. Nevertheless, it can still be better for concrete examples where W is small, as illustrated by the following exercise.

Exercise 6.2.5. Determine a maximal flow for the network of Figure 6.2 by always choosing an augmenting path of maximal capacity.

Exercise 6.2.6. Apply the algorithm of Edmonds and Karp to the network shown in Figure 6.13 (which is taken from [PaSt82]).

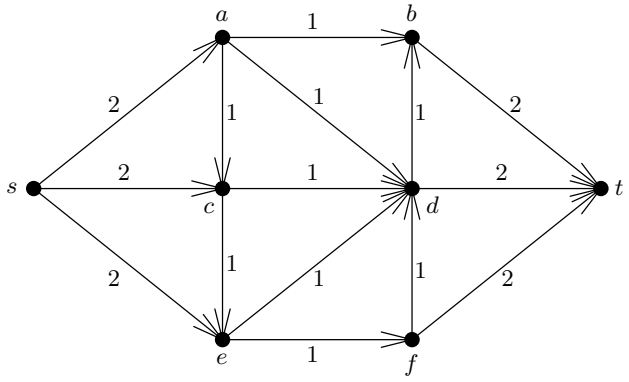


Fig. 6.13. A network

It can be shown that it is theoretically possible to obtain a maximal flow in a given network in at most $|E|$ iterations, and that this may even be achieved using augmenting paths which contain no backward edges; see [Law76, p. 119] and Exercise 7.1.3. However, this result is of not of any practical interest, as no algorithm is known which works with such paths only.

We conclude this section with three exercises showing that we can change several capacities in a given network and find solutions for the corresponding new problem without too much effort, if we know a solution of the original problem.

Exercise 6.2.7. Suppose we have determined a maximal flow for a flow network N using the algorithm of Edmonds and Karp, and realize afterwards that we used an incorrect capacity for some edge e. Discuss how we may use the solution of the original problem to solve the corrected problem.

Exercise 6.2.8. Change the capacity of the edge $e = ac$ in the network of Figure 6.2 to $c(e) = 8$, and then to $c(e) = 12$. How do these modifications change the value of a maximal flow? Give a maximal flow for each of these two cases.

Exercise 6.2.9. Change the network of Figure 6.2 as follows. The capacities of the edges ac and ad are increased to 12 and 16, respectively, and the edges de and ct are removed. Determine a maximal flow for the new network.

6.3 Auxiliary networks and phases

Let $N = (G, c, s, t)$ be a flow network with a flow f. We define a further flow network (G', c', s, t) as follows. G' has the same vertex set as G. For each edge $e = uv$ of G with $f(e) < c(e)$, there is an edge $e' = uv$ in G' with $c'(e') = c(e) - f(e)$; for each edge $e = uv$ with $f(e) \neq 0$, G' contains an edge $e'' = vu$ with $c'(e'') = f(e)$.

The labelling process in the algorithm of Ford and Fulkerson – as given in steps (6) to (9) for forward edges and in steps (10) to (13) for backward edges – uses only those edges e of G for which G' contains the edge e' or e''; an augmenting path with respect to f in G corresponds to a directed path from s to t in G'. Thus we may use G' to decide whether f is maximal and, if this is not the case, to find an augmenting path. One calls $N' = (G', c', s, t)$ the *auxiliary network* with respect to f. The next lemma should now be clear: it is just a translation of Theorem 6.1.3.

Lemma 6.3.1. *Let $N = (G, c, s, t)$ be a flow network with a flow f, and let N' be the corresponding auxiliary network. Then f is maximal if and only if t is not accessible from s in G'.* □

Example 6.3.2. Consider the flow $f = f_3$ of Example 6.2.3; see Figure 6.6. The corresponding auxiliary network is given in Figure 6.14.

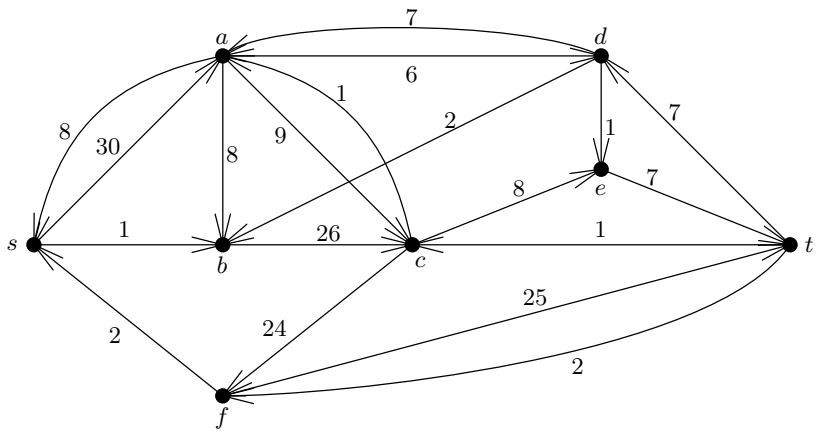


Fig. 6.14. Auxiliary network for Example 6.3.2

It is intuitively clear that a flow in N' can be used to augment f when constructing a maximal flow on N. The following two results make this idea more precise.

Lemma 6.3.3. *Let $N = (G, c, s, t)$ be a flow network with a flow f, and let N' be the corresponding auxiliary network. Moreover, let f' be a flow on N'. Then there exists a flow f'' of value $w(f'') = w(f) + w(f')$ on N.*

Proof. For each edge $e = uv$ of G, let $e' = uv$ and $e'' = vu$. If e' or e'' is not contained in N', we set $f'(e') = 0$ or $f'(e'') = 0$, respectively. We put $f'(e) = f'(e') - f'(e'')$;[2] then we may interpret f' as a (possibly non-admissible) *flow* on N: f' satisfies condition (F2), but not necessarily (F1). Obviously, the mapping f'' defined by

$$f''(e) \;=\; f(e) + f'(e') - f'(e'')$$

also satisfies condition (F2). Now the definition of N' shows that the conditions $0 \leq f'(e') \leq c(e) - f(e)$ and $0 \leq f'(e'') \leq f(e)$ hold for each edge e, so that f'' satisfies (F1) as well. Thus f'' is a flow, and clearly $w(f'') = w(f) + w(f')$. □

Theorem 6.3.4. *Let $N = (G, c, s, t)$ be a flow network with a flow f, and let N' be the corresponding auxiliary network. Denote the value of a maximal flow on N and on N' by w and w', respectively. Then $w = w' + w(f)$.*

Proof. By Lemma 6.3.3, $w \geq w' + w(f)$. Now let g be a maximal flow on N and define a flow g' on N' as follows: for each edge e of G, set

$$\begin{aligned} g'(e') \;&=\; g(e) - f(e) \quad && \text{if } g(e) > f(e); \\ g'(e'') \;&=\; f(e) - g(e) \quad && \text{if } g(e) < f(e). \end{aligned}$$

Note that e' and e'' really are edges of N' under the conditions given above and that their capacities are large enough to ensure the validity of (F1). For every other edge e^* in N', put $g'(e^*) = 0$. It is easy to check that g' is a flow of value $w(g') = w(g) - w(f)$ on N'. This shows $w' + w(f) \geq w$. □

Exercise 6.3.5. Give an alternative proof for Theorem 6.3.4 by proving that the capacity $c'(S, T)$ of a cut (S, T) in N' is equal to $c(S, T) - w(f)$.

Remark 6.3.6. Note that the graph G' may contain parallel edges even if G itself – as we always assume – does not. This phenomenon occurs when G contains antiparallel edges, say $d = uv$ and $e = vu$. Then G' contains the parallel edges d' and e'' with capacities $c'(d') = c(d) - f(d)$ and $c'(e'') = f(e)$, respectively. For the validity of the preceding proofs and the subsequent algorithms, it is important that parallel edges of G' are *not* identified (and

[2] Note that the minus sign in front of $f'(e'')$ is motivated by the fact that e' and e'' have opposite orientation.

their capacities not added). Indeed, if we identified the edges d' and e'' above into a new edge e^* with capacity $c'(e^*) = c(d) - f(d) + f(e)$, it would no longer be obvious how to distribute a flow value $f'(e^*)$ when defining f'' in the proof of Lemma 6.3.3: we would have to decide which part of $f'(e^*)$ should contribute to $f''(d)$ (with a plus sign) and which part to $f''(e)$ (with a minus sign). Of course, it would always be possible to arrange this in such a manner that a flow f'' satisfying the feasibility condition (F1) arises, but this would require some unpleasant case distinctions. For this reason, we allow G' to contain parallel edges.[3] However, when actually programming an algorithm using auxiliary networks, it might be worthwhile to identify parallel edges of G' and add the necessary case distinctions for distributing the flow on N' during the augmentation step. In addition, one should also simplify things then by *cancelling* flow on pairs of antiparallel edges in such a way that only one edge of such a pair carries a non-zero flow.

We have seen that it is possible to find a maximal flow for our original network N by finding appropriate flows in a series of auxiliary networks $N_1 = N'(f_0)$, $N_2 = N'(f_1), \ldots$ Note that the labelling process in the algorithm of Ford and Fulkerson amounts to constructing a new auxiliary network after each augmentation of the flow. Thus constructing the auxiliary networks explicitly cannot by itself result in a better algorithm; in order to achieve an improvement, we need to construct several augmenting paths within the same auxiliary network. We require a further definition. A flow f is called a *blocking flow* if every augmenting path with respect to f has to contain a backward edge. Trivially, any maximal flow is blocking as well. But the converse is false: for example, the flow f_8 of Example 6.2.3 displayed in Figure 6.11 is blocking, but not maximal.

There is yet another problem that needs to be addressed: the auxiliary networks constructed so far are still too big and complex. Indeed, the auxiliary network in Example 6.3.2 looks rather crowded. Hence we shall work with appropriate sub-networks instead. The main idea of the algorithm of Dinic [Din70] is to use not only an augmenting path of shortest length, but also to keep an appropriate small network $N''(f)$ basically unchanged – with just minor modifications – until every further augmenting path has to have larger length.

For better motivation, we return once more to the algorithm of Edmonds and Karp. Making step (5) of Algorithm 6.1.7 more precise in step (5′) of Theorem 6.2.1 ensures that the labelling process on the auxiliary network $N' = N'(f)$ runs as a BFS on G'; thus the labelling process divides G' into *levels* or *layers* of vertices having the same distance to s; see Section 3.3. As we are only interested in finding augmenting paths of shortest length, N' usually contains a lot of superfluous information: we may omit

[3] Alternatively, we could forbid G to contain antiparallel edges; this might be achieved, for instance, by always subdividing one edge of an antiparallel pair.

- all vertices $v \neq t$ with $d(s,v) \geq d(s,t)$ together with all edges incident with these vertices;
- all edges leading from some vertex in layer j to some vertex in a layer $i \leq j$.

The resulting network $N'' = N''(f) = (G'', c'', s, t)$ is called the *layered auxiliary network* with respect to f.[4] The name *layered* network comes from the fact that G'' is a *layered digraph*: the vertex set V of G'' is the disjoint union of subsets $V_0, \ldots, V_k$ and all edges of G'' have the form uv with $u \in V_i$ and $v \in V_{i+1}$ for some index i.

Example 6.3.7. Consider the flow $f = f_3$ in Example 6.2.3 and the corresponding auxiliary network N' displayed in Figure 6.14. The associated layered auxiliary network N'' is shown in Figure 6.15. Here the capacities are written in parentheses; the other numbers are the values of a blocking flow g on N'' which arises from the three augmenting paths displayed in Figures 6.6, 6.7, and 6.8. Note that all three paths of length four in Example 6.2.3 can now be seen in the considerably clearer network N''. Note that g is blocking but not maximal: the sequence (s, a, d, e, c, f, t) determines an augmenting path containing the backward edge ce.

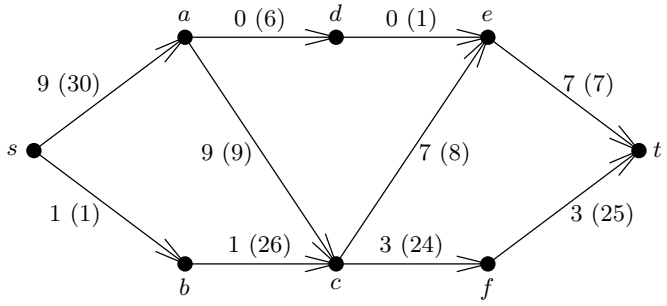


Fig. 6.15. Layered auxiliary network

We remark that even N'' might still contain superfluous elements, for example vertices from which t is not accessible. But as such vertices cannot be determined during the BFS used for constructing N'', we will not bother to find and remove them.

Exercise 6.3.8. How could vertices v in N'' from which t is not accessible be removed?

[4] Strictly speaking, both N' and N'' should probably only be called *networks* if t is accessible from s, that is, if f is not yet maximal – as this is part of the definition of flow networks. But it is more convenient to be a bit lax here.

Exercise 6.3.9. Draw N' and N'' for the flow f_7 displayed in Figure 6.10 and determine a blocking flow for N''.

We will treat two algorithms for determining maximal flows. Both algorithms can take a given flow f, construct a blocking flow g in the corresponding layered auxiliary network $N''(f)$, and then use g to augment f. Note that a flow f' of value $w(f')$ on $N''(f)$ may indeed be used to augment the given flow f to a flow of value $w(f) + w(f')$, as N'' is a sub-network of N'; hence we may apply Lemma 6.3.3 and the construction given in its proof.

Exercise 6.3.10. Show that Theorem 6.3.4 does not carry over to $N''(f)$.

Thus we begin with some starting flow f_0, usually the zero flow, construct a blocking flow g_0 in $N''(f_0)$, use this flow to augment f_0 to a flow f_1 of value $w(g_0)$, construct a blocking flow g_1 in $N''(f_1)$, and so on. The algorithm terminates when we reach a flow f_k for which the sink t is not accessible from s in $N''(f_k)$. Then t is not accessible from s in $N'(f_k)$ either; hence f_k is maximal, by Lemma 6.3.1. Each construction of a blocking flow g_i, together with the subsequent augmentation of f_i to f_{i+1}, is called a *phase* of the algorithm. We postpone the problem of determining blocking flows to the next section. Now we derive an estimate for the number of phases needed and write down an algorithm for constructing the layered auxiliary network.

Lemma 6.3.11. *Let $N = (G, c, s, t)$ be a flow network with a flow f, and let $N''(f)$ be the corresponding layered auxiliary network. Moreover, let g be a blocking flow on $N''(f)$, h the flow on N of value $w(f) + w(g)$ constructed from f and g as in Lemma 6.3.3, and $N''(h)$ the layered auxiliary network with respect to h. Then the distance from s to t in $N''(h)$ is larger than in $N''(f)$.*

Proof. We may view g as a flow on $N' := N'(f)$, by assigning value 0 to all edges contained in N', but not in $N'' := N''(f)$. Thus we may indeed construct a flow h on N from f and g as in Lemma 6.3.3. It is not difficult to see that $N'(h)$ is essentially – that is, up to identifying parallel edges – the auxiliary network for N' with respect to g; it follows that $N''(h)$ is the layered auxiliary network for N' with respect to g.[5] We leave the details to the reader.

As g is a blocking flow on N'', there is no augmenting path from s to t in N'' consisting of forward edges only. Hence each augmenting path in N' with respect to g has to contain a backward edge or one of those edges which were omitted during the construction of N''. In both cases, the length of this path must be larger than the distance from s to t in N''. Thus the distance from s to t in the layered auxiliary network for N' with respect to g – that is, in $N''(h)$ – is indeed larger than the corresponding distance in N''. □

[5] The analogous claim for $N'' = N''(f)$ instead of $N'(f)$ does not hold, as Exercise 6.3.13 will show.

Corollary 6.3.12. *Let N be a flow network. Then the construction of a maximal flow on N needs at most $|V| - 1$ phases.*

Proof. Let $f_0, f_1, \ldots, f_k$ be the sequence of flows constructed during the algorithm. Lemma 6.3.11 implies that the distance from s to t in $N''(f_k)$ is at least k larger than that in $N''(f_0)$. Thus the number of phases can be at most $|V| - 1$. □

Exercise 6.3.13. Let f be the flow f_3 in Example 6.2.3 and g the blocking flow on $N''(f)$ in Example 6.3.7. Draw the layered auxiliary networks with respect to g on $N'(f)$ and on $N''(f)$. What does the flow h determined by f and g on N look like? Convince yourself that $N''(h)$ is indeed equal to the layered auxiliary network with respect to g on $N'(f)$.

The following procedure for constructing the layered auxiliary network $N'' = N''(f)$ corresponds to the labelling process in the algorithm of Ford and Fulkerson with step (5) replaced by (5′) – as in Theorem 6.2.1. During the execution of the BFS, the procedure orders the vertices in layers and omits superfluous vertices and edges, as described in the definition of N''. The Boolean variable max is assigned the value *true* when f becomes maximal (that is, when t is no longer accessible from s); otherwise, it has value *false*. The variable $d + 1$ gives the number of layers of N''.

Algorithm 6.3.14. Let $N = (G, c, s, t)$ be a flow network with a flow f.

Procedure AUXNET$(N, f; N'', \max, d)$

(1) $i \leftarrow 0$, $V_0 \leftarrow \{s\}$, $E'' \leftarrow \emptyset$, $V'' \leftarrow V_0$;
(2) **repeat**
(3) $\quad i \leftarrow i + 1$, $V_i \leftarrow \emptyset$;
(4) $\quad$ **for** $v \in V_{i-1}$ **do**
(5) $\quad\quad$ **for** $e \in \{e \in E : e^- = v\}$ **do**
(6) $\quad\quad\quad$ **if** $u = e^+ \notin V''$ **and** $f(e) < c(e)$
(7) $\quad\quad\quad$ **then** $e' \leftarrow vu$, $E'' \leftarrow E \cup \{e'\}$, $V_i \leftarrow V_i \cup \{u\}$;
$\quad\quad\quad\quad c''(e') \leftarrow c(e) - f(e)$ **fi**
(8) $\quad\quad$ **od**
(9) $\quad\quad$ **for** $e \in \{e \in E : e^+ = v\}$ **do**
(10) $\quad\quad\quad$ **if** $u = e^- \notin V''$ **and** $f(e) \neq 0$
(11) $\quad\quad\quad$ **then** $e'' \leftarrow vu$, $E'' \leftarrow E \cup \{e''\}$, $V_i \leftarrow V_i \cup \{u\}$;
$\quad\quad\quad\quad c''(e'') \leftarrow f(e)$ **fi**
(12) $\quad\quad$ **od**
(13) $\quad$ **od**
(14) $\quad$ **if** $t \in V_i$ **then** remove all vertices $v \neq t$ together with all edges e satisfying $e^+ = v$ from V_i **fi**
(15) $\quad V'' \leftarrow V'' \cup V_i$
(16) **until** $t \in V''$ **or** $V_i = \emptyset$;
(17) **if** $t \in V''$ **then** max $\leftarrow$ false; $d \leftarrow i$ **else** max $\leftarrow$ true **fi**

We leave it to the reader to give a formal proof for the following lemma.

Lemma 6.3.15. *Algorithm 6.3.14 constructs the layered auxiliary network* $N'' = N''(f) = (G'', c'', s, t)$ *on* $G'' = (V'', E'')$ *with complexity* $O(|E|)$. □

In the next section, we will provide two methods for constructing a blocking flow g on N''. Let us assume for the moment that we already know such a procedure BLOCKFLOW$(N''; g)$. Then we want to use g for augmenting f. The following procedure performs this task; it uses the construction given in the proof of Lemma 6.3.3. Note that N'' never contains both e' and e''.

Algorithm 6.3.16. Let $N = (G, c, s, t)$ be a given flow network with a flow f, and suppose that we have already constructed $N'' = N''(f)$ and a blocking flow g.

Procedure AUGMENT$(f, g; f)$

(1) **for** $e \in E$ **do**
(2) **if** $e' \in E''$ **then** $f(e) \leftarrow f(e) + g(e')$ **fi**
(3) **if** $e'' \in E''$ **then** $f(e) \leftarrow f(e) - g(e'')$ **fi**
(4) **od**

We can now write down an algorithm for determining a maximal flow:

Algorithm 6.3.17. Let $N = (G, c, s, t)$ be a flow network.

Procedure MAXFLOW$(N; f)$

(1) **for** $e \in E$ **do** $f(e) \leftarrow 0$ **od**
(2) **repeat**
(3) AUXNET$(N, f; N'', \max, d)$;
(4) **if** max = false
(5) **then** BLOCKFLOW$(N''; g)$; AUGMENT$(f, g; f)$ **fi**
(6) **until** max = true

Remark 6.3.18. The only part which is still missing in Algorithm 6.3.17 is a specific procedure BLOCKFLOW for determining a blocking flow g on N''. Note that each phase of Algorithm 6.3.17 has complexity at least $O(|E|)$, because AUGMENT has this complexity. It is quite obvious that BLOCKFLOW will also have complexity at least $O(|E|)$; in fact, the known algorithms have even larger complexity. Let us denote the complexity of BLOCKFLOW by $k(N)$. Then MAXFLOW has a complexity of $O(|V|k(N))$, since there are at most $O(|V|)$ phases, by Corollary 6.3.12.

Exercise 6.3.19. Modify Algorithm 6.3.17 in such a way that it finds a minimal cut (S, T) as well.

6.4 Constructing blocking flows

In this section, we fill in the gap left in Algorithm 6.3.17 by presenting two algorithms for constructing blocking flows. The first of these is due to Dinic [Din70]. The Dinic algorithm constructs, starting with the zero flow, augmenting paths of length d in the layered auxiliary network N'' (where $d+1$ denotes the number of layers) and uses them to change the flow g until t is no longer accessible from s; then g is a blocking flow. Compared to the algorithm of Edmonds and Karp, it has two advantages. First, using $N'' = N''(f)$ means that we consider only augmenting paths without any backward edges, since a path containing a backward edge has length at least $d + 2$. Second, when we update the input data after an augmentation of the current flow g on N'', we only have to decrease the capacities of the edges contained in the respective augmenting path and omit vertices and edges that are no longer needed. In particular, we do *not* have to do the entire labelling process again.

Algorithm 6.4.1. Let $N = (G, c, s, t)$ be a layered flow network with layers $V_0, \ldots, V_d$, where all capacities are positive.

Procedure BLOCKFLOW$(N; g)$

(1) **for** $e \in E$ **do** $g(e) \leftarrow 0$ **od**
(2) **repeat**
(3) $v \leftarrow t$, $a \leftarrow \infty$;
(4) **for** $i = d$ **downto** 1 **do**
(5) choose some edge $e_i = uv$;
(6) $a \leftarrow \min\{c(e_i), a\}$, $v \leftarrow u$
(7) **od**
(8) **for** $i = 1$ **to** d **do**
(9) $g(e_i) \leftarrow g(e_i) + a$, $c(e_i) \leftarrow c(e_i) - a$;
(10) **if** $c(e_i) = 0$ **then** omit e_i from E **fi**
(11) **od**
(12) **for** $i = 1$ **to** d **do**
(13) **for** $v \in V_i$ **do**
(14) **if** $d_{\text{in}}(v) = 0$
(15) **then** omit v and all edges e with $e^- = v$ **fi**
(16) **od**
(17) **od**
(18) **until** $t \notin V_d$

Theorem 6.4.2. *Algorithm 6.4.1 determines a blocking flow on N with complexity $O(|V||E|)$.*

Proof. By definition of a layered auxiliary network, each vertex is accessible from s at the beginning of the algorithm. Thus there always exists an edge e_i with end vertex v which can be chosen in step (5), no matter which edges

$e_d, \ldots, e_{i+1}$ were chosen before. Hence the algorithm constructs a directed path $P = (e_1, \ldots, e_d)$ from s to t. At the end of the loop (4) to (7), the variable a contains the capacity a of P, namely $a = \min\{c(e_i) : i = 1, \ldots, d\}$. In steps (8) to (11), the flow constructed so far (in the first iteration, the zero flow) is increased by a units along P, and the capacities of the edges $e_1, \ldots, e_d$ are decreased accordingly. Edges whose capacity is decreased to 0 cannot appear on any further augmenting path and are therefore discarded. At the end of the loop (8) to (11), we have reached t and augmented the flow g. Before executing a further iteration of (4) to (11), we have to check whether t is still accessible from s. Even more, we need to ensure that every vertex is still accessible from s in the modified layered network. This task is performed by the loop (12) to (17). Using induction on i, one may show that this loop removes exactly those vertices which are not accessible from s as well as all edges beginning in these vertices. If t is still contained in N after this loop has ended, we may augment g again so that we repeat the entire process. Finally, the algorithm terminates after at most $|E|$ iterations, since at least one edge is removed during each augmentation; at the very latest, t can no longer be in V_d when all the edges have been removed. Obviously, each iteration (4) to (17) has complexity $O(|V|)$; this gives the desired overall complexity of $O(|V||E|)$. □

Using Remark 6.3.18, we immediately obtain the following result due to Dinic.

Corollary 6.4.3. *Assume that we use the procedure BLOCKFLOW of Algorithm 6.4.1 in Algorithm 6.3.17. Then the resulting algorithm calculates a maximal flow on a given flow network N with complexity $O(|V|^2|E|)$.* □

Note that the algorithm of Dinic has a complexity of $O(|V|^4)$ for dense graphs, whereas the algorithm of Edmonds and Karp needs $O(|V|^5)$ steps in this case. Using another, more involved, method for constructing blocking flows, we may reduce the complexity to $O(|V|^3)$ for arbitrary graphs. But first, let us work out an example for the algorithm of Dinic.

Example 6.4.4. Consider again the flow $f = f_3$ in Example 6.2.3. The corresponding layered auxiliary network N'' was displayed in Figure 6.15. We apply Algorithm 6.4.1 to N''. In step (5), let us always choose the edge uv for which u is first in alphabetical order, so that the algorithm becomes deterministic. Initially, it constructs the path $s — a — c — e — t$ with capacity 7. The corresponding flow g_1 is shown in Figure 6.16; the numbers in parentheses give the new capacities (which were changed when the flow was changed). The edge et, which is drawn broken, is removed during this first iteration.

In the second iteration, we obtain the path $s — a — c — f — t$ in the network of Figure 6.16; it has capacity two. Figure 6.17 shows the new network with the new flow g_2. Note that the edge ac has been removed.

Finally, using the network in Figure 6.17, the third iteration constructs the path $s — b — c — f — t$ with capacity one, and we obtain the flow g_3

displayed in Figure 6.18. During this iteration, the algorithm removes first the edge sb, the vertex b, and the edge bc; then the vertex c, and the edges ce and cf; the vertex f, and the edge ft; and finally t itself; see Figure 6.18. Hence g_3 is a blocking flow – actually, the blocking flow previously displayed in Figure 6.15.

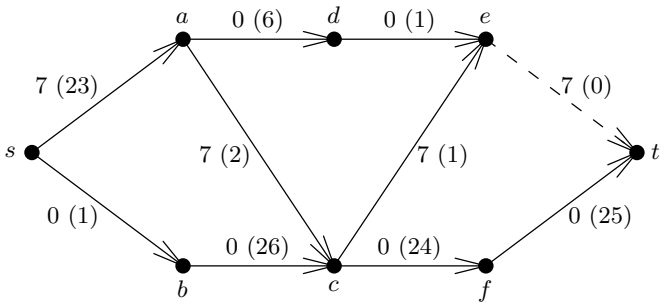


Fig. 6.16. $w(g_1) = 7$

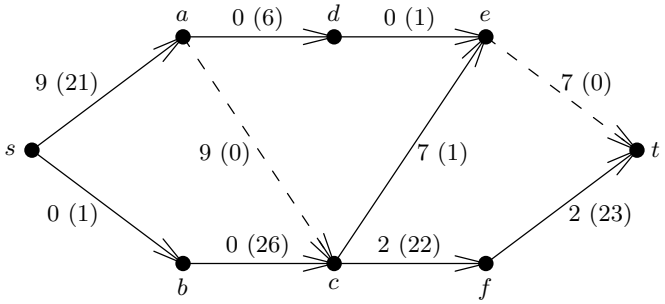


Fig. 6.17. $w(g_2) = 9$

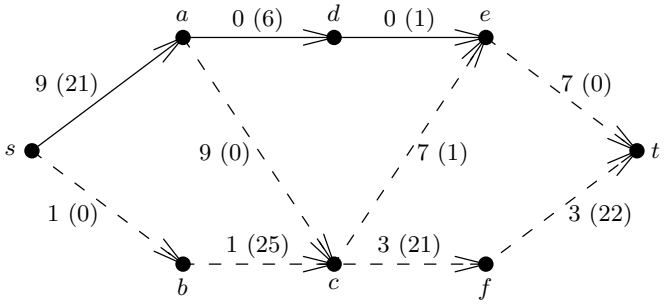


Fig. 6.18. Blocking flow g_3 with $w(g_3) = 10$

Exercise 6.4.5. Use Algorithm 6.4.1 to determine a blocking flow on the layered network shown in Figure 6.19 [SyDK83].

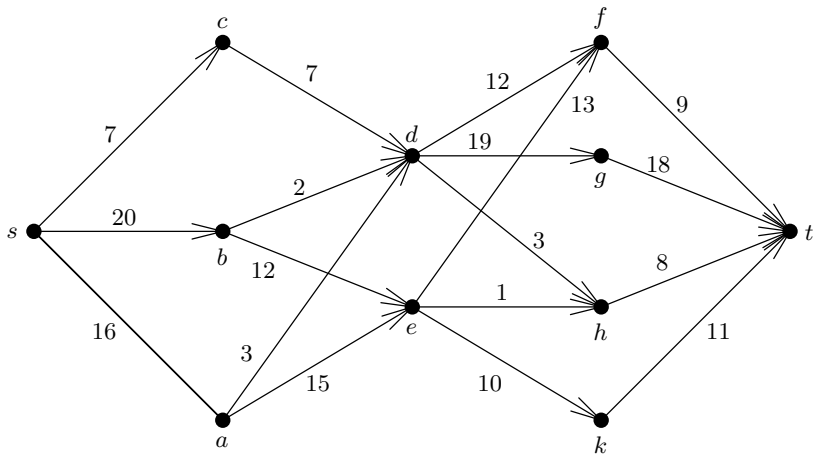


Fig. 6.19. A layered network

We now turn to a completely different method for constructing blocking flows, which is due to Malhotra, Kumar and Mahashwari [MaKM78] and has complexity $O(|V|^2)$. This algorithm does not use augmenting paths and tries instead to push as big a flow as possible through the network. We need some notation. Let $N = (G, c, s, t)$ be a layered flow network. For each vertex v, the *flow potential* $p(v)$ is defined by

$$p(v) \;=\; \min\left\{\sum_{e^-=v} c(e),\ \sum_{e^+=v} c(e)\right\};$$

thus $p(v)$ is the maximal amount of flow which could possibly pass through v. A vertex u is called a *minimal vertex* – and its flow potential the *minimal potential* – if $p(u) \leq p(v)$ holds for all vertices v.

Intuitively, it should be possible to construct a flow g of value $w(g) = p(u)$ by pushing the flow from u forward to t and pulling it back from u to s. This is the main idea of the following algorithm for constructing a blocking flow on N.

Algorithm 6.4.6. Let $N = (G, c, s, t)$ be a layered flow network with layers $V_1, \ldots V_d$, where all capacities are positive.

Procedure BLOCKMKM($N; g$)

(1) **for** $e \in E$ **do** $g(e) \leftarrow 0$ **od**
(2) **for** $v \in V$ **do**

(3) **if** $v = t$ **then** $p^-(v) \leftarrow \infty$ **else** $p^-(v) \leftarrow \sum_{e^-=v} c(e)$ **fi**
(4) **if** $v = s$ **then** $p^+(v) \leftarrow \infty$ **else** $p^+(v) \leftarrow \sum_{e^+=v} c(e)$ **fi**
(5) **od**
(6) **repeat**
(7) **for** $v \in V$ **do** $p(v) \leftarrow \min\{p^+(v), p^-(v)\}$ **od**
(8) choose a minimal vertex w;
(9) PUSH$(w, p(w))$
(10) PULL$(w, p(w))$
(11) **while** there exists v with $p^+(v) = 0$ or $p^-(v) = 0$ **do**
(12) **for** $e \in \{e \in E : e^- = v\}$ **do**
(13) $u \leftarrow e^+$, $p^+(u) \leftarrow p^+(u) - c(e)$
(14) remove e from E
(15) **od**
(16) **for** $e \in \{e \in E : e^+ = v\}$ **do**
(17) $u \leftarrow e^-$, $p^-(u) \leftarrow p^-(u) - c(e)$
(18) remove e from E
(19) **od**
(20) remove v from V
(21) **od**
(22) **until** $s \notin V$ or $t \notin V$

Here, PUSH is the following procedure for *pushing a flow of value* $p(w)$ *to* t:

Procedure PUSH(y, k)

(1) let Q be a queue with single element y;
(2) **for** $u \in V$ **do** $b(u) \leftarrow 0$ **od**
(3) $b(y) \leftarrow k$;
(4) **repeat**
(5) remove the first element v from Q;
(6) **while** $v \neq t$ **and** $b(v) \neq 0$ **do**
(7) choose an edge $e = vu$;
(8) $m \leftarrow \min\{c(e), b(v)\}$;
(9) $c(e) \leftarrow c(e) - m$, $g(e) \leftarrow g(e) + m$;
(10) $p^+(u) \leftarrow p^+(u) - m$, $b(u) \leftarrow b(u) + m$;
(11) $p^-(v) \leftarrow p^-(v) - m$, $b(v) \leftarrow b(v) - m$;
(12) append u to Q;
(13) **if** $c(e) = 0$ **then** remove e from E **fi**
(14) **od**
(15) **until** $Q = \emptyset$

The procedure PULL for *pulling a flow of value* $p(w)$ *to* s is defined in an analogous manner; we leave this task to the reader.

Theorem 6.4.7. *Algorithm 6.4.6 constructs a blocking flow g on N with complexity $O(|V|^2)$.*

Proof. We claim first that an edge e is removed from E only if there exists no augmenting path containing e and consisting of forward edges only. This is clear if e is removed in step (14) or (18): then either $p^+(v) = 0$ or $p^-(v) = 0$ (where $e^- = v$ or $e^+ = v$, respectively) so that no augmenting path containing v and consisting of forward edges only can exist. If e is removed in step (13) during a call of the procedure PUSH, we have $c(e) = 0$ at this point; because of step (9) in PUSH, this means that $g(e)$ has reached its original capacity $c(e)$ so that e cannot be used any longer as a forward edge. A similar argument applies if e is removed during a call of PULL. As each iteration of BLOCKMKM removes edges and decreases capacities, an edge which can no longer be used as a forward edge with respect to g when it is removed cannot be used as a forward edge at a later point either. Hence, there cannot exist any augmenting path consisting of forward edges only at the end of BLOCKMKM, when s or t have been removed. This shows that g is blocking; of course, it still remains to check that g is a flow in the first place.

We now show that g is indeed a flow, by using induction on the number of iterations of the **repeat**-loop (6) to (22). Initially, g is the zero flow. Now suppose that g is a flow at a certain point of the algorithm (after the i-th iteration, say). All vertices v which cannot be used any more – that is, vertices into which no flow can enter or from which no flow can emerge any more – are removed during the **while**-loop (11) to (21), together with all edges incident with these vertices. During the next iteration – that is, after the flow potentials have been brought up to date in step (7) – the algorithm chooses a vertex w with minimal potential $p(w)$; here $p(w) \neq 0$, since otherwise w would have been removed before during the **while**-loop. Next, we have to check that the procedure PUSH$(w, p(w))$ really generates a flow of value $p(w)$ from the source w to the sink t. As Q is a queue, the vertices u in PUSH are treated as in a BFS on the layers $V_k, V_{k+1}, \ldots, V_d$, where $w \in V_k$. During the first iteration of the **repeat**-loop of PUSH, we have $v = w$ and $b(v) = p(w)$; here $b(v)$ contains the value of the flow which has to flow out of v. During the **while**-loop, the flow of value $b(v)$ is distributed among the edges vu with tail v. Note that the capacity of an edge vu is always used entirely, unless $b(v) < c(e)$. In step (9), the capacity of vu is reduced – in most cases, to 0, so that vu will be removed in step (13) – and the value of the flow is increased accordingly. Then we decrease the value $b(v)$ of the flow which still has to leave v via other edges accordingly in step (11), and increase $b(u)$ accordingly in step (10); also the flow potentials are updated by the appropriate amount. In this way the required value of the flow $b(v)$ is distributed among the vertices of the next layer; as we chose w to be a vertex of minimal potential, we always have $b(v) \leq p(w) \leq p(v)$, and hence it is indeed possible to distribute the flow. At the end of procedure PUSH, the flow of value $p(w)$ has reached t, since $V_d = \{t\}$. An analogous argument shows that the subsequent call of the

procedure PULL$(w, p(w))$ yields a flow of value $p(w)$ from the source s to the sink w; of course, PULL performs the actual construction in the opposite direction. We leave the details to the reader. Hence g will indeed be a flow from s to t after both procedures have been called.

Each iteration of the **repeat**-loop of BLOCKMKM removes at least one vertex, since the flow potential of the minimal vertex w is decreased to 0 during PUSH and PULL; hence the algorithm terminates after at most $|V|-1$ iterations. We now need an estimate for the number of operations involving edges. Initializing p^+ and p^- in (3) and (4) takes $O(|E|)$ steps altogether. As an edge e can be removed only once, e appears at most once during the **for**-loops (12) to (19) or in step (13) of PUSH or PULL. For each vertex v treated during PUSH or PULL, there is at most one edge starting in v which still has a capacity $\neq 0$ left after it has been processed – that is, which has not been removed. As PUSH and PULL are called at most $|V|-1$ times each, we need at most $O(|V|^2)$ steps for treating these special edges. But $O(|V|^2)$ dominates $O(|E|)$; hence the overall number of operations needed for treating the edges is $O(|V|^2)$. It is easy to see that all other operations of the algorithm need at most $O(|V|^2)$ steps as well, so that we obtain the desired overall complexity of $O(|V|^2)$. □

The algorithm arising from Algorithm 6.3.17 by replacing BLOCKFLOW with BLOCKMKM is called the *MKM-algorithm*. As explained in Remark 6.3.18, Theorem 6.4.7 implies the following result.

Theorem 6.4.8. *The MKM-algorithm constructs with complexity $O(|V|^3)$ a maximal flow for a given flow network N .* □

Example 6.4.9. Consider again the layered auxiliary network of Example 6.3.7. Here the flow potentials are as follows: $p(s) = 31$, $p(a) = 15$, $p(b) = 1$, $p(c) = 32$, $p(d) = 1$, $p(e) = 7$, $p(f) = 24$, $p(t) = 32$. Let us choose b as minimal vertex in step (8). After the first iteration, we have the flow g_1 shown in Figure 6.20; the vertex b as well as the edges sb and bc have been removed.

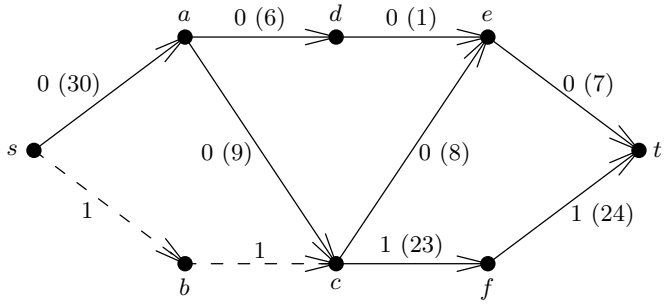


Fig. 6.20. $w(g_1) = 1$

Next, we have flow potentials $p(s) = 30$, $p(a) = 15$, $p(c) = 9$, $p(d) = 1$, $p(e) = 7$, $p(f) = 23$, $p(t) = 31$, so that d is the unique minimal vertex. After the second iteration, we have constructed the flow g_2 in Figure 6.21; also, d, ad, and de have been removed.

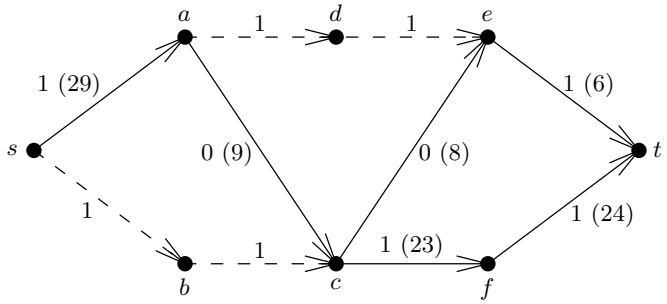


Fig. 6.21. $w(g_2) = 2$

In the following iteration, $p(s) = 29$, $p(a) = 9$, $p(c) = 9$, $p(e) = 6$, $p(f) = 23$ and $p(t) = 30$. Hence the vertex e is minimal and we obtain the flow g_3 shown in Figure 6.22; note that e, ce, and et have been removed.

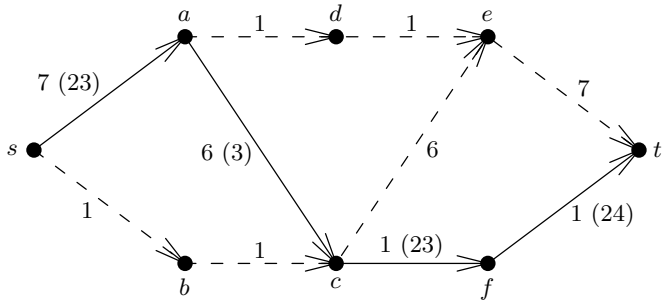


Fig. 6.22. $w(g_3) = 8$

Now the flow potentials are $p(s) = 23$, $p(a) = 3$, $p(c) = 3$, $p(f) = 23$, $p(t) = 24$. We select the minimal vertex a and construct the flow g_4 in Figure 6.23, a blocking flow with value $w(g_4) = 11$; all remaining elements of the network have been removed. Note that g_4 differs from the blocking flow constructed in Example 6.4.4.

Exercise 6.4.10. Use Algorithm 6.4.6 to find a blocking flow for the layered auxiliary network of Exercise 6.4.5.

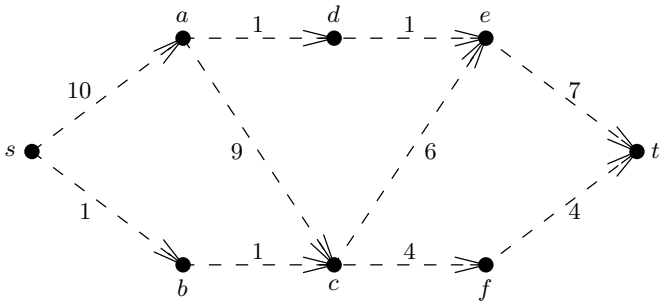


Fig. 6.23. $w(g_4) = 11$

We have now seen three classical algorithms for constructing maximal flows. Note that these algorithms have quite different complexities for the case of dense graphs, that is, for $|E| = O(|V|^2)$. As this case shows, the MKM-algorithm is superior to the other two algorithms; however, it is clearly also considerably more involved.

Further algorithms with complexity $O(|V|^3)$ are given in [Kar74], [Tar84], and [GoTa88]; the last of these papers also contains an algorithm of complexity $O(|V||E| \log(|V|^2/|E|))$. We shall present the algorithm of Goldberg and Tarjan in Section 6.6. The approach of Goldberg and Tarjan can also be used to solve the *parametrized flow problem*, where the capacities of the edges incident with s and t depend monotonically on some real parameter; interestingly, the complexity will only change by a constant factor; see [GaGT89]. A modification of the algorithm of Goldberg and Tarjan suggested in [ChMa89] results in a complexity $O(|V|^2|E|^{1/2})$; we shall include this result in Section 6.6 as well.

An algorithm with complexity $O(|V||E| \log |V|)$ is given in [Sle80]; see also [SlTa83]. In the paper [AhOr89], there is an algorithm of complexity $O(|V||E| + |V|^2 \log U)$, where U denotes the maximum of all capacities $c(e)$ occurring in the problem. This result may be improved somewhat: the term $\log U$ can be replaced by $(\log U)^{1/2}$; see [AhOT89].

A probabilistic algorithm was proposed by [ChHa89]. Later, several authors gave deterministic variants of this algorithm; see [Alo90], [KiRT94], and [ChHa95]. For instance, for graphs satisfying $|V|(\log |V|)^2 \leq |E| \leq |V|^{5/3} \log |V|$, one obtains a complexity of only $O(|V||E|)$. An algorithm with complexity $O(|V|^3/\log |V|)$ can be found in [ChHM96].

Let us also mention three good surveys dealing with some of the more recent algorithms: [GoTT90] and [AhMO89, AhMO91]. For an extensive treatment of flow theory and related topics, we refer to the monograph [AhMO93].

A further new idea for solving the max-flow problem emerged in a paper by Karger [Kar99], who proceeds by computing approximately minimal cuts and uses these to compute a maximum flow, thus reversing the usual approach. He

shows that his algorithm has an improved complexity with high probability; however, this seems not to be of practical interest (yet).

A completely different approach to the problem of finding a maximal flow is to use the well-known simplex algorithm from linear programming and specialize it to treat flow networks; the resulting algorithm actually works for a more general problem. It is called the *network simplex algorithm* and is of eminent practical interest; we will devote the entire Chapter 11 to this topic.

For planar graphs, one may construct a maximal flow with complexity $O(|V|^{\frac{3}{2}} \log |V|)$; see [JoVe82]. If s and t are in the same face of G, even a complexity of $O(|V| \log |V|)$ suffices; see [ItSh79].

In the undirected case – that is, in *symmetric flow networks*, see Section 12.1 – the max flow problem can be solved with complexity $O(|V| \log^2 |V|)$; see [HaJo85]. For flow networks on bipartite graphs, fast algorithms can be found in [GuMF87] and in [AhOST94]; these algorithms are particularly interesting if one of the two components of the bipartition is very small compared to the other component.

Finally, let us mention some references discussing the practical efficiency of various flow algorithms: [Che80], [Gal81], [Ima83], [GoGr88], [DeMe89], and [AhKMO92]. There are also several relevant papers in the collection [JoMcG93].

6.5 Zero-one flows

In this section, we consider a special case occurring in many combinatorial applications of flow theory[6], namely integral flows which take the values 0 and 1 only. In this special case, the complexity estimates we have obtained so far can be improved considerably; it will suffice to use the algorithm of Dinic for this purpose. We need some terminology. A 0-1-*network* is a flow network $N = (G, c, s, t)$ for which all capacities $c(e)$ are restricted to the values 0 and 1. A flow f on a network N is called a 0-1-*flow* if f takes values 0 and 1 only. We begin with the following important lemma taken from [EvTa75].

Lemma 6.5.1. *Let $N = (G, c, s, t)$ be a 0-1-network. Then*

$$d(s,t) \;\leq\; \frac{2|V|}{\sqrt{M}}, \tag{6.5}$$

where M is the maximal value of a flow on N. If, in addition, each vertex v of N except for s and t satisfies at least one of the two conditions $d_{\text{in}}(v) \leq 1$ and $d_{\text{out}}(v) \leq 1$, then even

$$d(s,t) \;\leq\; 1 + \frac{|V|}{M}. \tag{6.6}$$

[6] We will discuss a wealth of such applications in Chapter 7.

Proof. Denote the maximal distance from s in N by D, and let V_i be the set of all vertices $v \in V$ with $d(s,v) = i$ (for $i = 0, \ldots, D$). Obviously,

$$(S_i, T_i) \;=\; (V_0 \cup V_1 \cup \ldots \cup V_i,\, V_{i+1} \cup \ldots \cup V_D)$$

is a cut, for each $i < d(s,t)$. As every edge e with $e^- \in S_i$ and $e^+ \in T_i$ satisfies $e^- \in V_i$ and $e^+ \in V_{i+1}$ and as N is a 0-1-network, Lemma 6.1.2 implies

$$M \;\leq\; c(S_i, T_i) \;\leq\; |V_i| \times |V_{i+1}| \quad \text{for } i = 0, \ldots, d(s,t) - 1.$$

Thus at least one of the two values $|V_i|$ or $|V_{i+1}|$ cannot be smaller than $\sqrt{M}$. Hence at least half of the layers V_i with $i \leq d(s,t)$ contain $\sqrt{M}$ or more vertices. This yields

$$d(s,t)\frac{\sqrt{M}}{2} \;\leq\; |V_0| + \ldots + |V_{d(s,t)}| \;\leq\; |V|,$$

and hence (6.5) holds. Now assume that N satisfies the additional condition stated in the assertion. Then the flow through any given vertex cannot exceed one, and we get the stronger inequality

$$M \;\leq\; |V_i| \quad \text{for } i = 1, \ldots, d(s,t) - 1.$$

Now

$$M(d(s,t) - 1) \;\leq\; |V_1| + \ldots + |V_{d(s,t)-1}| \;\leq\; |V|,$$

proving (6.6). □

Using estimates which are a little more accurate, (6.5) can be improved to $d(s,t) \leq |V|/\sqrt{M}$. We will not need this improvement for the complexity statements which we will establish later in this section; the reader might derive the stronger bound as an exercise.

Lemma 6.5.2. *Let $N = (G, c, s, t)$ be a layered 0-1-network. Then the algorithm of Dinic can be used to determine a blocking flow g on N with complexity $O(|E|)$.*

Proof. The reader may easily check that the following modification of Algorithm 6.4.1 determines a blocking flow g on a given 0-1-network N.

Procedure BLOCK01FLOW(N, g)

(1) $L \leftarrow \emptyset$;
(2) **for** $v \in V$ **do** $ind(v) \leftarrow 0$ **od**
(3) **for** $e \in E$ **do** $g(e) \leftarrow 0$; $ind(e^+) \leftarrow ind(e^+) + 1$ **od**
(4) **repeat**
(5) $\quad v \leftarrow t$;
(6) $\quad$ **for** $i = d$ **downto** 1 **do**
(7) $\quad\quad$ choose an edge $e = uv$ and remove e from E;
(8) $\quad\quad$ $ind(v) \leftarrow ind(v) - 1$; $g(e) \leftarrow 1$;

```
 (9)        if ind(v) = 0
(10)        then append v to L;
(11)           while L ≠ ∅ do
(12)              remove the first vertex w from L;
(13)              for {e ∈ E : e⁻ = w} do
(14)                 remove e from E; ind(e⁺) ← ind(e⁺) − 1;
(15)                 if ind(e⁺) = 0 then append e⁺ to L fi
(16)              od
(17)           od
(18)        fi
(19)        v ← u;
(20)     od
(21)  until ind(t) = 0
```

Obviously, each edge e is treated – and then removed – at most once during the **repeat**-loop in this procedure, so that the complexity of BLOCK01FLOW is $O(|E|)$. □

Theorem 6.5.3. *Let $N = (G, c, s, t)$ be a* 0-1*-network. Then the algorithm of Dinic can be used to compute a maximal* 0-1*-flow on N with complexity $O(|V|^{2/3}|E|)$.*

Proof. In view of Lemma 6.5.2, it suffices to show that the algorithm of Dinic needs only $O(|V|^{2/3})$ phases when it runs on a 0-1-network. Let us denote the maximal value of a flow on N by M. As the value of the flow is increased by at least 1 during each phase of the algorithm, the assertion is trivial whenever $M \leq |V|^{2/3}$; thus we may suppose $M > |V|^{2/3}$. Consider the uniquely determined phase where the value of the flow is increased to a value exceeding $M - |V|^{2/3}$, and let f be the 0-1-flow on N which the algorithm had constructed in the immediately preceding phase. Then $w(f) \leq M - |V|^{2/3}$, and therefore the value M' of a maximal flow on $N'(f)$ is given by $M' = M - w(f) \geq |V|^{2/3}$, by Theorem 6.3.4. Obviously, N' is likewise a 0-1-network, so that the distance $d(s,t)$ from s to t in $N'(f)$ satisfies the inequality

$$d(s,t) \;\leq\; \frac{2|V|}{\sqrt{M'}} \;\leq\; 2|V|^{2/3},$$

by Lemma 6.5.1. Now Lemma 6.3.11 guarantees that the distance between s and t in the corresponding auxiliary network increases in each phase, and hence the construction of f can have taken at most $2|V|^{2/3}$ phases. By our choice of f, we reach a flow value exceeding $M - |V|^{2/3}$ in the next phase, so that at most $|V|^{2/3}$ phases are necessary to increase the value of the flow step by step until it reaches M. Hence the number of phases is indeed at most $O(|V|^{2/3})$. □

In exactly the same manner, we get a further improvement of the complexity provided that the 0-1-network N satisfies the additional condition of

Lemma 6.5.1 and hence the stronger inequality (6.6). Of course, this time the threshold M used in the argument should be chosen as $|V|^{1/2}$; also note that the 0-1-network $N'(f)$ likewise satisfies the additional hypothesis in Lemma 6.5.1. We leave the details to the reader and just state the final result.

Theorem 6.5.4. *Let $N = (G, c, s, t)$ be a 0-1-network. If each vertex $v \neq s, t$ of N satisfies at least one of the two conditions $d_{\text{in}}(v) \leq 1$ and $d_{\text{out}}(v) \leq 1$, then the algorithm of Dinic can be used to determine a maximal 0-1-flow on N with complexity $O(|V|^{1/2}|E|)$.* □

We close this section with an example and two exercises outlining some applications of 0-1-flows; they touch some very interesting questions which we will study in considerably more detail later. We shall also present several further applications of 0-1-flows in Chapter 7.

Example 6.5.5. Given a bipartite graph $G = (S \dot\cup T, E)$, we seek a matching of maximal cardinality in G; see Exercise 5.1.5. Let us show that this problem is equivalent to finding a maximal 0-1-flow on an appropriate flow network. We define a digraph H by adding two new vertices s and t to the vertex set $S \cup T$ of G. The edges of H are all the sx for $x \in S$, all the xy for which $\{x, y\}$ is an edge of G with $x \in S$, and all the yt for $y \in T$. All edges are assigned capacity 1. This defines a 0-1-network N.

Note that the edges $\{x_i, y_i\}$ $(i = 1, \ldots, k)$ of an arbitrary matching of G induce a flow of value k on N: put $f(e) = 1$ for all edges $e = sx_i$, $e = x_i y_i$, and $e = y_i t$ (for $i = 1, \ldots, k$). Conversely, a 0-1-flow of value k yields a matching consisting of k edges: select the k edges of the type xy which actually carry a non-zero flow.

The problem transformation just described directly leads to an effective algorithm for determining maximal matchings in bipartite graphs: by Theorem 6.5.4, a maximal 0-1-flow on N can be determined with complexity $O(|V|^{1/2}|E|)$; thus the complexity is at most $O(|V|^{5/2})$. More precisely, we may use the following algorithm:[7]

Procedure MATCH$(G; K)$

(1) let s and t be two new vertices; $V' \leftarrow S \cup T \cup \{s, t\}$
(2) **for** $e \in E$ **do if** $e = \{x, y\}$ with $x \in S$ **then** $e' = xy$ **fi od**
(3) $E' \leftarrow \{sx : x \in S\} \cup \{e' : e \in E\} \cup \{yt : y \in T\}$
(4) **for** $e \in E'$ **do** $c(e) \leftarrow 1$ **od**
(5) $H \leftarrow (V', E')$, $N \leftarrow (H, c, s, t)$, $K \leftarrow \emptyset$
(6) MAXFLOW$(N; f)$
(7) **for** $e \in E$ **do if** $f(e') = 1$ **then** $K \leftarrow K \cup \{e\}$ **fi od**

[7] Note that backward edges may occur during the algorithm. This does not interfere with the final solution, because $c(sv) = c(wt) = 1$ holds for all v and w, so that at most one edge of the form vw incident with v or w, respectively, can carry a non-zero flow.

Of course, in order to achieve the desired complexity $O(|V|^{1/2}|E|)$, MAXFLOW in step (6) should be specified according to the results of the present section, using BLOCK01FLOW in Algorithm 6.3.17.

The method for finding a maximal matching described in Example 6.5.5 is basically due to Hopcroft and Karp [HoKa73], who used a rather different presentation; later, it was noticed by Evan and Tarjan [EvTa75] that this method may be viewed as a special case of the MAXFLOW-algorithm. We will meet maximal matchings quite often in this book: the bipartite case will be treated in Section 7.2, and the general case will be studied in Chapters 13 and 14.

Exercise 6.5.6. A prom is attended by m girls and n boys. We want to arrange a dance where as many couples as possible should participate, but only couples who have known each other before are allowed. Formulate this task as a graph theoretical problem.

Exercise 6.5.7. Let $G = (S \dot\cup T, E)$ be a bipartite graph. Show that the set system $(S, \mathbf{S})$ defined by

$$\mathbf{S} = \big\{X \subset S\colon \text{ there exists a matching } M \text{ with } X = \{e^- : e \in M\}\big\}$$

is a matroid; here e^- denotes that vertex incident with e which lies in S. Hint: Use the interpretation via a network given in Exercise 6.5.5 for a constructive proof of condition (3) in Theorem 5.2.1.

The result in Exercise 6.5.7 becomes even more interesting when seen in contrast to the fact that the set $\mathbf{M}$ of all matchings does not form a matroid on E; see Exercise 5.1.5.

6.6 The algorithm of Goldberg and Tarjan

This final section of Chapter 6 is devoted to a more recent algorithm for finding maximal flows which is due to Goldberg and Tarjan [GoTa88]. The algorithms we have presented so far construct a maximal flow – usually starting with the zero flow – by augmenting the flow iteratively, either along a single augmenting path or in phases where blocking flows in appropriate auxiliary networks are determined.

The algorithm of Goldberg and Tarjan is based on a completely different concept: it uses *preflows*. These are mappings for which *flow excess* is allowed: the amount of flow entering a vertex may be larger than the amount of flow leaving it. This preflow property is maintained throughout the algorithm; it is only at the very end of the algorithm that the preflow becomes a flow – which is then already maximal.

The main idea of the algorithm is to push flow from vertices with excess flow toward the sink t, using paths which are not necessarily shortest paths

from s to t, but merely current estimates for such paths. Of course, it might occur that excess flow cannot be pushed forward from some vertex v; in this case, it has to be sent back to the source on a suitable path. The choice of all these paths is controlled by a certain labelling function on the vertex set. We will soon make all this precise. Altogether, the algorithm will be quite intuitive and comparatively simple to analyze. Moreover, it needs a complexity of only $O(|V|^3)$, without using any special tricks. By applying more complicated data structures, it can even be made considerably faster, as we have already noted at the end of Section 6.4.

Following [GoTa88], we define flows in this section in a formally different – although, of course, equivalent – way; this notation from [Sle80] will simplify the presentation of the algorithm. First, it is convenient to consider c and f also as functions from $V \times V$ to $\mathbb{R}$. Thus we do not distinguish between $f(e)$ and $f(u,v)$, where $e = uv$ is an edge of G; we put $f(u,v) = 0$ whenever uv is not an edge of G; and similarly for c. Then we drop the condition that flows have to be nonnegative, and define a flow $f : V \times V \to \mathbb{R}$ by the following requirements:

(1) $f(v,w) \leq c(v,w)$ for all $(v,w) \in V \times V$

(2) $f(v,w) = -f(w,v)$ for all $(v,w) \in V \times V$

(3) $\sum_{u \in V} f(u,v) = 0$ for all $v \in V \setminus \{s,t\}$.

The anti-symmetry condition (2) makes sure that only one of the two edges in a pair vw and wv of antiparallel edges in G may carry a positive amount of flow.[8] Condition (2) also simplifies the formal description in one important respect: we will not have to make a distinction between forward and backward edges anymore. Moreover, the formulation of the flow conservation condition (3) is easier. The definition of the value of a flow becomes a little easier, too:

$$w(f) = \sum_{v \in V} f(v,t).$$

For an intuitive interpretation of flows in the new sense, the reader should consider only the nonnegative part of the flow function: this part is a flow as originally defined in Section 6.1. As an exercise, the reader is asked to use the antisymmetry of f to check that condition (3) is equivalent to the earlier condition (F2).

Now we define a *preflow* as a mapping $f : V \times V \to \mathbb{R}$ satisfying conditions (1) and (2) above and the following weaker version of condition (3):

(3′) $\sum_{u \in V} f(u,v) \geq 0$ for all $v \in V \setminus \{s\}$.

[8] In view of condition (2) we have to assume that G is a *symmetric* digraph: if vw is an edge, wv must also be an edge of G. As noted earlier, there is no need for positive amounts of flow on two antiparallel edges: we could simply cancel flow whenever such a situation occurs.

Using the intuitive interpretation of flows, condition (3′) means that the amount of flow entering a vertex $v \neq s$ no longer has to equal the amount leaving v; it suffices if the in-flow is always at least as large as the out-flow. The value

$$e(v) \;=\; \sum_{u \in V} f(u,v)$$

is called the *flow excess* of the preflow f in v.

As mentioned before, the algorithm of Goldberg and Tarjan tries to push flow excess from some vertex v with $e(v) > 0$ forward towards t. We first need to specify which edges may be used for pushing flow. This amounts to defining an auxiliary network, similar to the one used in the classical algorithms; however, the algorithm itself does not involve an explicit construction of this network. Given a preflow f, let us define the *residual capacity* $r_f : V \times V \to \mathbb{R}$ as follows:

$$r_f(v,w) \;=\; c(v,w) - f(v,w).$$

If an edge vw satisfies $r_f(v,w) > 0$, we may move some flow through this edge; such an edge is called a *residual edge*. In our intuitive interpretation, this corresponds to two possible cases. Either the edge vw is a forward edge which is not yet saturated: $0 \leq f(v,w) < c(v,w)$; or it is a backward edge, that is, the antiparallel edge wv is non-void: $0 < f(w,v) \leq c(w,v)$, and hence $f(v,w) = -f(w,v) < 0 \leq c(v,w)$. The *residual graph* with respect to f is defined as

$$G_f = (V, E_f), \quad \text{where} \quad E_f = \{vw \in E\colon\, r_f(v,w) > 0\}.$$

As the intuitive interpretation shows, G_f really corresponds to the auxiliary network $N'(f)$ used in the classical algorithms. Now we may also introduce the labelling function mentioned before. A mapping $d : V \to \mathbb{N}_0 \cup \{\infty\}$ is called a *valid labelling* with respect to a given preflow f if the following two conditions hold:

(4) $d(s) = |V|$ and $d(t) = 0$;

(5) $d(v) \leq d(w) + 1$ for all $vw \in E_f$.

The algorithm of [GoTa88] starts with some suitable preflow and a corresponding valid labelling. Usually, one saturates all edges emanating from s, and puts $d(s) = |V|$ and $d(v) = 0$ for all $v \in V \setminus \{s\}$. More precisely, the initial preflow is given by $f(s,v) = -f(v,s) = c(s,v)$ for all $v \neq s$ and $f(v,w) = 0$ for all $v, w \neq s$.

Then the algorithm executes a series of operations which we will specify later. These operations change either the preflow f (by pushing the largest possible amount of flow along a suitable residual edge) or the labelling d (by raising the label of a suitable vertex); in both cases, the labelling will always remain valid. As mentioned before, d is used to estimate shortest paths in the corresponding residual graph; see Lemma 6.6.4. In particular, $d(v)$ is always

a lower bound for the distance from v to t in G_f provided that $d(v) < |V|$; and if $d(v) \geq |V|$, then t is not accessible from v, and $d(v) - |V|$ is a lower bound for the distance from v to s in G_f. The algorithm terminates as soon as the preflow has become a flow (which is then actually a maximal flow).

We need one more notion to be able to write down the algorithm in its generic form. A vertex v is called *active* provided that $v \neq s, t$; $e(v) > 0$; and $d(v) < \infty$.

Algorithm 6.6.1. Let $N = (G, c, s, t)$ be a flow network on a symmetric digraph, where $c\colon V \times V \to \mathbb{R}_0^+$; that is, for $(v, w) \notin E$ we have $c(v, w) = 0$.

Procedure GOLDBERG($N; f$)

(1) **for** $(v, w) \in (V \setminus \{s\}) \times (V \setminus \{s\})$ **do** $f(v, w) \leftarrow 0$; $r_f(v, w) \leftarrow c(v, w)$ **od**
(2) $d(s) \leftarrow |V|$;
(3) **for** $v \in V \setminus \{s\}$ **do**
(4) $\quad f(s, v) \leftarrow c(s, v)$; $r_f(s, v) \leftarrow 0$;
(5) $\quad f(v, s) \leftarrow -c(s, v)$; $r_f(v, s) \leftarrow c(v, s) + c(s, v)$;
(6) $\quad d(v) \leftarrow 0$;
(7) $\quad e(v) \leftarrow c(s, v)$
(8) **od**
(9) **while** there exists an active vertex v **do**
(10) $\quad$ choose an active vertex v and execute an admissible operation
(11) **od**

In (10), one of the following operations may be used, provided that it is admissible:

Procedure PUSH($N, f, v, w; f$)

(1) $\delta \leftarrow \min\,(e(v), r_f(v, w))$;
(2) $f(v, w) \leftarrow f(v, w) + \delta$; $f(w, v) \leftarrow f(w, v) - \delta$;
(3) $r_f(v, w) \leftarrow r_f(v, w) - \delta$; $r_f(w, v) \leftarrow r_f(w, v) + \delta$;
(4) $e(v) \leftarrow e(v) - \delta$; $e(w) \leftarrow e(w) + \delta$.

The procedure PUSH($N, f, v, w; f$) is *admissible* provided that v is active, $r_f(v, w) > 0$, and $d(v) = d(w) + 1$.

Procedure RELABEL($N, f, v, d; d$)

(1) $d(v) \leftarrow \min\,\{d(w) + 1\colon r_f(v, w) > 0\}$;

The procedure RELABEL($N, f, v, d; d$) is *admissible* provided that v is active and $r_f(v, w) > 0$ always implies $d(v) \leq d(w)$.[9]

Let us look more closely at the conditions for admissibility. If we want to push some flow along an edge vw, three conditions are required. Two of these requirements are clear: the start vertex v has to be active, so that there is

[9] The minimum in (1) is defined to be ∞ if there does not exist any w with $r_f(v, w) > 0$. However, we will see that this case cannot occur.

positive flow excess $e(v)$ available which we might move; and vw has to be a residual edge, so that it has capacity left for additional flow. It is also not surprising that we then push along vw as much flow as possible, namely the smaller of the two amounts $e(v)$ and $r_f(v,w)$.

The crucial requirement is the third one, namely $d(v) = d(w)+1$. Thus we are only allowed to push along residual edges vw for which $d(v)$ is exactly one unit larger than $d(w)$, that is, for which $d(v)$ takes its maximum permissible value; see (5) above. We may visualize this rule by thinking of water cascading down a series of terraces of different height, with the height corresponding to the labels. Obviously, water will flow down, and condition (5) has the effect of restricting the layout of the terraces so that the water may flow down only one level in each step.

Now assume that we are in an active vertex v – so that some water is left which wants to flow out – and that none of the residual edges leaving v satisfies the third requirement. In our watery analogy, v would be a sort of local sink: v is locally on the lowest possible level, and thus the water is trapped in v. It is precisely in such a situation that the RELABEL-operation becomes admissible: we miraculously raise v to a level which is just one unit higher than that of the lowest neighbor w of v in G_f; then a PUSH becomes permissible, that is, (some of) the water previously trapped in v can flow down to w. Of course, these remarks in no way constitute a proof of correctness; nevertheless, they might help to obtain a feeling for the strategy behind the Goldberg-Tarjan algorithm.

Now we turn to the formal proof which proceeds via a series of auxiliary results. This will allow us to show that Algorithm 6.6.1 constructs a maximal flow on N in finitely many steps, no matter in which order we select the active vertices and the admissible operations. This is in remarkable contrast to the situation for the algorithm of Ford and Fulkerson; recall the discussion in Section 6.1. To get better estimates for the complexity, however, we will have to specify appropriate strategies for the choices to be made.

We begin by showing that the algorithm is correct under the assumption that it terminates at all. Afterwards, we will estimate the maximal number of admissible operations executed during the **while**-loop and use this result to show that the algorithm really is finite. Our first lemma is just a simple but important observation; it states a result which we have already emphasized in our informal discussion.

Lemma 6.6.2. *Let f be a preflow on N, d a valid labelling on V with respect to f, and v an active vertex. Then either a* PUSH*-operation or a* RELABEL*-operation is admissible for v.*

Proof. As d is valid, we have $d(v) \leq d(w) + 1$ for all w with $r_f(v,w) > 0$. If PUSH(v,w) is not admissible for any w, we must even have $d(v) \leq d(w)$ for all w with $r_f(v,w) > 0$, as d takes only integer values. But then RELABEL is admissible. □

Lemma 6.6.3. *During the execution of Algorithm 6.6.1, f always is a preflow and d always is a valid labelling (with respect to f).*

Proof. We use induction on the number k of admissible operations already executed. The assertion holds for the induction basis $k = 0$: obviously, f is initialized as a preflow in steps (4) and (5); and the labelling d defined in (2) and (6) is valid for f, since $d(v) = 0$ for $v \neq s$ and since all edges sv have been saturated in step (4); also, the residual capacities and the flow excesses are clearly initialized correctly in steps (4), (5), and (7).

For the induction step, suppose that the assertion holds after k operations have been executed. Assume first that the next operation is a PUSH(v, w). It is easy to check that f remains a preflow, and that the residual capacities and the flow excesses are updated correctly. Note that the labels are kept unchanged, and that vw and wv are the only edges for which f has changed. Hence we only need to worry about these two edges in order to show that d is still valid. By definition, $vw \in E_f$ before the PUSH. Now vw might be removed from the residual graph G_f (which happens if it is saturated by the PUSH); but then the labelling stays valid trivially. Now consider the antiparallel edge wv. If this edge already is in G_f, there is nothing to show. Thus assume that wv is added to G_f by the PUSH; again, d stays valid, since the admissibility conditions for the PUSH(v, w) require $d(w) = d(v) - 1$.

It remains to consider the case where the next operation is a RELABEL(v). Then the admissibility requirement is $d(v) \leq d(w)$ for all vertices w with $r_f(v, w) > 0$. As $d(v)$ is increased to the minimum of all the $d(w) + 1$, the condition $d(v) \leq d(w) + 1$ holds for all w with $r_f(v, w) > 0$ after this change; all other labels remain unchanged, so that the new labelling d is still valid for f. □

Our next lemma is a simple but very useful observation. As mentioned before, the valid labelling d allows us to estimate distances in the corresponding residual graph:

Lemma 6.6.4. *Let f be a preflow on N, let d be a valid labelling with respect to f, and let v and w be two vertices of N such that w is accessible from v in the residual graph G_f. Then $d(v) - d(w)$ is a lower bound for the distance of v and w in G_f:*

$$d(v) - d(w) \;\leq\; d(v, w).$$

Proof. Let

$$P\colon\; v = v_0 \text{ — } v_1 \text{ — } \ldots \text{ — } v_k = w$$

be a shortest path from v to w in G_f. Since d is a valid labelling,

$$d(v_i) \;\leq\; d(v_{i+1}) + 1 \quad \text{for } i = 0, \ldots, k-1.$$

As P has length $k = d(v, w)$, we obtain the desired inequality. □

Corollary 6.6.5. *Let f be a preflow on N and d a valid labelling with respect to f. Then t is not accessible from s in the residual graph G_f.*

Proof. Assume otherwise. Then $d(s) \leq d(t) + d(s,t)$, by Lemma 6.6.4. But this contradicts $d(s) = |V|$, $d(t) = 0$, and $d(s,t) \leq |V| - 1$. □

Theorem 6.6.6. *If Algorithm 6.6.1 terminates with all labels finite, then the preflow f constructed is in fact a maximal flow on N.*

Proof. By Lemma 6.6.2, the algorithm can only terminate when there are no more active vertices. As all labels are finite by hypothesis, $e(v) = 0$ has to hold for each vertex $v \neq s, t$; hence the preflow constructed by the final operation is indeed a flow on N. By Corollary 6.6.5, there is no path from s to t in G_f, so that there is no augmenting path from s to t with respect to f. Now the assertion follows from Theorem 6.1.3. □

It remains to show that the algorithm indeed terminates and that the labels stay finite throughout. We need several further lemmas.

Lemma 6.6.7. *Let f be a preflow on N. If v is a vertex with positive flow excess $e(v)$, then s is accessible from v in G_f.*

Proof. We denote the set of vertices accessible from v in G_f (via a directed path) by S, and put $T := V \setminus S$. Then $f(u,w) \leq 0$ for all vertices u, w with $u \in T$ and $w \in S$, since

$$0 \;=\; r_f(w,u) \;=\; c(w,u) - f(w,u) \;\geq\; 0 + f(u,w).$$

Using the antisymmetry of f, we get

$$\begin{aligned} \sum_{w \in S} e(w) \;&=\; \sum_{u \in V, w \in S} f(u,w) \\ &=\; \sum_{u \in T, w \in S} f(u,w) + \sum_{u,w \in S} f(u,w) \\ &=\; \sum_{u \in T, w \in S} f(u,w) \;\leq\; 0. \end{aligned}$$

Now the definition of a preflow requires $e(w) \geq 0$ for all $w \neq s$. But $e(v) > 0$, and hence $\sum_{w \in S} e(w) \leq 0$ implies $s \in S$. □

Lemma 6.6.8. *Throughout Algorithm 6.6.1, $d(v) \leq 2|V| - 1$ for all $v \in V$.*

Proof. Obviously, the assertion holds after the initialization phase in steps (1) to (8). The label $d(v)$ of a vertex v can only be changed by an operation RELABEL(v), and such an operation is admissible only if v is active. In particular, $v \neq s, t$, so that the claim is trivial for s and t; moreover, $e(v) > 0$.

By Lemma 6.6.7, s is accessible from v in the residual graph G_f. Now Lemma 6.6.4 gives

$$d(v) \;\le\; d(s) + d(v,s) \;\le\; d(s) + |V| - 1 \;=\; 2|V| - 1. \qquad \square$$

Lemma 6.6.9. *During the execution of Algorithm 6.6.1, at most* $2|V| - 1$ RELABEL*-operations occur for any given vertex* $v \neq s, t$*. Hence the total number of* RELABEL*-operations is at most* $(2|V| - 1)(|V| - 2) < 2|V|^2$*.*

Proof. Each RELABEL(v) increases $d(v)$. Since $d(v)$ is bounded by $2|V| - 1$ throughout the entire algorithm (see Lemma 6.6.8), the assertion follows. $\square$

It is more difficult to estimate the number of PUSH-operations. We need to distinguish two cases: a PUSH(v, w) will be called a *saturating* PUSH if $r_f(v, w) = 0$ holds afterwards (that is, for $\delta = r_f(v, w)$ in step (1) of the PUSH), and a *non-saturating* PUSH otherwise.

Lemma 6.6.10. *During the execution of Algorithm 6.6.1, fewer than* $|V||E|$ *saturating* PUSH*-operations occur.*

Proof. By definition, any PUSH(v, w) requires $vw \in E_f$ and $d(v) = d(w) + 1$. If the PUSH is saturating, a further PUSH(v, w) can only occur after an intermediate PUSH(w, v), since we have $r_f(v, w) = 0$ after the saturating PUSH(v, w). Note that no PUSH(w, v) is admissible before the labels have been changed in such a way that $d(w) = d(v)+1$ holds; hence $d(w)$ must have been increased by at least 2 units before the PUSH(w, v). Similarly, no further PUSH(v, w) can become admissible before $d(v)$ has also been increased by at least 2 units. In particular, $d(v) + d(w)$ has to increase by at least 4 units between any two consecutive saturating PUSH(v, w)-operations.

On the other hand, $d(v) + d(w) \ge 1$ holds as soon as the first PUSH from v to w or from w to v is executed. Moreover, $d(v), d(w) \le 2|V| - 1$ throughout the algorithm, by Lemma 6.6.8; hence $d(v) + d(w) \le 4|V| - 2$ holds when the last PUSH-operation involving v and w occurs. Therefore there are at most $|V| - 1$ saturating PUSH(v, w)-operations, so that the the total number of saturating PUSH-operations cannot exceed $(|V| - 1)|E|$. $\square$

Lemma 6.6.11. *During the execution of Algorithm 6.6.1, there are at most* $2|V|^2|E|$ *non-saturating* PUSH*-operations.*

Proof. Let us introduce the *potential*

$$\Phi \;=\; \sum_{v \text{ active}} d(v)$$

and investigate its development during the course of Algorithm 6.6.1. After the initialization phase, $\Phi = 0$; and at the end of the algorithm, we have $\Phi = 0$ again.

Note that any non-saturating PUSH(v, w) decreases Φ by at least one unit: because $r_f(v, w) > e(v)$, the vertex v becomes inactive so that Φ is decreased by $d(v)$ units; and even if the vertex w has become active due to the PUSH, Φ is increased again by only $d(w) = d(v) - 1$ units, as the PUSH must have been admissible. Similarly, any saturating PUSH(v, w) increases[10] Φ by at most $2|V| - 1$, since the label of the vertex w – which might again have become active due to this PUSH – satisfies $d(w) \leq 2|V| - 1$, by Lemma 6.6.8.

Let us put together what these observations imply for the entire algorithm. The saturating PUSH-operations increase Φ by at most $(2|V| - 1)|V||E|$ units altogether, by Lemma 6.6.10; and the RELABEL-operations increase Φ by at most $(2|V|-1)(|V|-2)$ units, by Lemma 6.6.8. Clearly, the value by which Φ is increased over the entire algorithm must be the same as the value by which it is decreased again. As this happens for the non-saturating PUSH-operations, we obtain an upper bound of $(2|V| - 1)(|V||E| + |V| - 2)$ for the total number of non-saturating PUSH-operations. Now the bound in the assertion follows easily, using that G is connected. □

The preceding lemmas combine to give the desired result:

Theorem 6.6.12. *Algorithm 6.6.1 terminates after at most $O(|V|^2|E|)$ admissible operations (with a maximal flow).* □

Exercise 6.6.13. Let f be a maximal flow on a flow network N with n vertices which has been obtained from Algorithm 6.6.1, and let d be the associated valid labelling. Show the existence of some i with $1 \leq i \leq n - 1$ such that $S = \{v \in V\colon d(v) > i\}$ and $T = \{w \in V\colon d(w) < i\}$ form a minimal cut.

The precise complexity of Algorithm 6.6.1 depends both on the way the admissible operations are implemented and on the order in which they are applied in the **while**-loop. In any case, the running time will be polynomial. We shall treat two variants which lead to particularly good results; they differ only in the manner in which they select the active vertex in step (10). Both variants use the obvious strategy not to change the active vertex v unnecessarily, but to stick with v until

- either $e(v) = 0$,
- or all edges incident with v have already been used for a PUSH(v, w), as far as this is possible, and a RELABEL(v) has occurred afterwards.

To implement this strategy, we use incidence lists. For each vertex v, there always is a distinguished *current edge* in its incidence list A_v (which may be implemented via a pointer). Initially, this edge is just the first edge of A_v; thus we assume A_v to have a fixed order. In the following algorithm, the active vertices are selected according to the rule *first in first out* – which explains its name.

[10] Note that a saturating PUSH may actually even *decrease* Φ.

Algorithm 6.6.14 (FIFO preflow push algorithm). Let $N = (G, c, s, t)$ be a flow network, where G is a symmetric digraph given by incidence lists A_v. Moreover, Q denotes a queue and *rel* a Boolean variable.

Procedure FIFOFLOW$(N; f)$

(1) **for** $(v, w) \in (V \setminus \{s\}) \times (V \setminus \{s\})$ **do** $f(v, w) \leftarrow 0$; $r_f(v, w) \leftarrow c(v, w)$ **od**
(2) $d(s) \leftarrow |V|$; $Q \leftarrow \emptyset$;
(3) **for** $v \in V \setminus \{s\}$ **do**
(4) $\quad f(s, v) \leftarrow c(s, v)$; $r_f(s, v) \leftarrow 0$;
(5) $\quad f(v, s) \leftarrow -c(s, v)$; $r_f(v, s) \leftarrow c(v, s) + c(s, v)$;
(6) $\quad d(v) \leftarrow 0$; $e(v) \leftarrow c(s, v)$;
(7) $\quad$ make the first edge in A_v the current edge;
(8) $\quad$ **if** $e(v) > 0$ **and** $v \neq t$ **then** append v to Q **fi**
(9) **od**
(10) **while** $Q \neq \emptyset$ **do**
(11) $\quad$ remove the first vertex v from Q; rel $\leftarrow$ false;
(12) $\quad$ **repeat**
(13) $\quad\quad$ let vw be the current edge in A_v;
(14) $\quad\quad$ **if** $r_f(v, w) > 0$ **and** $d(v) = d(w) + 1$
(15) $\quad\quad$ **then** PUSH$(N, f, v, w; f)$;
(16) $\quad\quad\quad$ **if** $w \notin Q$ **and** $w \neq s, t$ **then** append w to Q **fi**
(17) $\quad\quad$ **fi**
(18) $\quad\quad$ **if** $e(v) > 0$ **then**
(19) $\quad\quad\quad$ **if** vw is not the last edge in A_v
(20) $\quad\quad\quad$ **then** choose the next edge in A_v as current edge
(21) $\quad\quad\quad$ **else** RELABEL$(N, f, v, d; d)$; rel $\leftarrow$ true;
(22) $\quad\quad\quad\quad$ make the first edge in A_v the current edge
(23) $\quad\quad\quad$ **fi**
(24) $\quad\quad$ **fi**
(25) $\quad$ **until** $e(v) = 0$ **or** rel = true;
(26) $\quad$ **if** $e(v) > 0$ **then** append v to Q **fi**
(27) **od**

The reader may show that Algorithm 6.6.14 is indeed a special case of Algorithm 6.6.1; this amounts to checking that RELABEL(v) is called only when no PUSH along an edge starting in v is admissible. By Theorem 6.6.12, the algorithm terminates with a maximal flow on N. The following result giving its complexity is due to Goldberg and Tarjan [GoTa88].

Theorem 6.6.15. *Algorithm 6.6.14 determines with complexity $O(|V|^3)$ a maximal flow on N.*

Proof. Obviously, the initialization phase in steps (1) to (9) has complexity $O(|E|)$. In order to analyze the complexity of the **while**-loop, we divide the

course of the algorithm into *phases*.[11] Phase 1 consists of the execution of the **repeat**-loop for those vertices which were originally appended to Q, that is, when Q was initialized in step (8). If phase i is already defined, phase $i+1$ consists of the execution of the **repeat**-loop for those vertices which were appended to Q during phase i. We claim that there are at most $O(|V|^2)$ phases.

By Lemma 6.6.9, there are at most $O(|V|^2)$ phases involving a RELABEL. It remains to establish the same bound for phases without a RELABEL. For this purpose, we take the same approach as in the proof of Lemma 6.6.11: we define a potential and investigate its development during the course of the algorithm. This time, we let Φ be the maximum value of the labels $d(v)$, taken over all active vertices v. Let us consider how Φ changes during a phase not involving any RELABEL-operations. Then, for each active vertex v, excess flow is moved to vertices w with label $d(v)-1$ until we finally reach $e(v)=0$, so that v ceases to be active. Of course, Φ cannot be increased by these operations; and at the end of such a phase – when all originally active vertices v have become inactive – Φ has actually decreased by at least one unit. Hence, if Φ remains unchanged or increases during a phase, at least one RELABEL-operation must occur during this phase; we already noted that there are at most $O(|V|^2)$ phases of this type. As $\Phi = 0$ holds at the beginning as well as at the end of the algorithm, at most $O(|V|^2)$ decreases of Φ can occur. Hence there are indeed at most $O(|V|^2)$ phases not involving a RELABEL.

We can now estimate the number of steps required for all PUSH-operations; note that an individual PUSH needs only $O(1)$ steps. Hence we want to show that there are only $O(|V|^3)$ PUSH-operations. In view of Lemma 6.6.10, it suffices to consider non-saturating PUSH-operations. Note that the **repeat**-loop for a vertex v is aborted as soon as a non-saturating PUSH(v,w) occurs; see step (25). Clearly, at most $O(|V|)$ vertices v are investigated during a phase, so that there are at most $O(|V|)$ non-saturating PUSH-operations during each phase. Now our result on the number of phases gives the assertion.

It remains to estimate how often each edge is examined during the **while**-loop. Consider the edges starting in a specified vertex v. During a **repeat**-loop involving v, the current edge e runs through (part of) the incidence list A_v of v. More precisely, the pointer is moved to the next edge whenever treating e leaves v with flow excess $e(v) > 0$; and the pointer is returned to the first edge only when a RELABEL(v) occurs. By Lemma 6.6.9, each vertex v is relabeled at most $2|V|-1$ times, so that the incidence list A_v of v is examined only $O(|V|)$ times during the entire algorithm. (Note that this estimate also includes the complexity of the RELABEL-operations: each RELABEL(v) also amounts to looking through all edges in A_v.) Hence we obtain altogether $O(|V||A_v|)$ examinations of the edges starting in v; summing

[11] In the original literature, the phases are called *passes over* Q, which seems somewhat misleading.

this over all vertices shows that the edge examinations and the RELABEL-operations only contribute $O(|V||E|)$ to the complexity of the algorithm. □

Examples which show that the FIFO-algorithm might indeed need $O(|V|^3)$ steps are provided in [ChMa89].

We now turn to our second variation of Algorithm 6.6.1. This time, we always choose an active vertex which has the maximal label among all the active vertices. To implement this strategy, we use a priority queue with priority function d instead of an ordinary queue. This variant was likewise suggested by Goldberg and Tarjan [GoTa88].

Algorithm 6.6.16 (highest label preflow push algorithm). Let $N = (G, c, s, t)$ be a flow network, where G is a symmetric digraph given by incidence lists A_v. Moreover, let Q be a priority queue with priority function d, and *rel* a Boolean variable.

Procedure HLFLOW($N; f$)

```
(1) for (v,w) ∈ (V \ {s}) × (V \ {s}) do f(v,w) ← 0; r_f(v,w) ← c(v,w) od
(2) d(s) ← |V|; Q ← ∅;
(3) for v ∈ V \ {s} do
(4)       f(s,v) ← c(s,v); r_f(s,v) ← 0;
(5)       f(v,s) ← −c(s,v); r_f(v,s) ← c(v,s) + c(s,v);
(6)       d(v) ← 0; e(v) ← c(s,v);
(7)       make the first edge in A_v the current edge;
(8)       if e(v) > 0 and v ≠ t then insert v into Q with priority d(v) fi
(9) od
(10) while Q ≠ ∅ do
(11)      remove a vertex v of highest priority d(v) from Q; rel ← false;
(12)      repeat
(13)           let vw be the current edge in A_v;
(14)           if r_f(v,w) > 0 and d(v) = d(w) + 1 then
(15)                PUSH(N, f, v, w; f);
(16)                if w ∉ Q and w ≠ s,t then insert w into Q
                    with priority d(w) fi
(17)           fi
(18)           if e(v) > 0 then
(19)                if vw is not the last edge in A_v
(20)                then choose the next edge in A_v as current edge;
(21)                else RELABEL(N, f, v, d; d); rel ← true;
(22)                     make the first edge in A_v the current edge;
(23)                fi
(24)           fi
(25)      until e(v) = 0 or rel = true;
(26)      if e(v) > 0 then insert v into Q with priority d(v) fi
(27) od
```

Goldberg and Tarjan proved that Algorithm 6.6.16 has a complexity of at most $O(|V|^3)$; this estimate was improved by Cheriyan and Maheshwari [ChMa89] as follows.

Theorem 6.6.17. *Algorithm 6.6.16 determines with complexity* $O(|V|^2|E|^{1/2})$ *a maximal flow on* N.

Proof.[12] As in the proof of Theorem 6.6.15, the main problem is to establish the necessary bound for the number of non-saturating PUSH-operations; all other estimates can be done as before. Note here that $O(|V||E|)$ – that is, the bound for the saturating PUSH-operations provided by Lemma 6.6.10 – is indeed dominated by $O(|V|^2|E|^{1/2})$.

As in the proof of Theorem 6.6.15, we divide the algorithm into phases; but this time, a phase consists of all operations occurring between two consecutive RELABEL-operations. The *length* l_i of the i-th phase is defined as the difference between the values of $d_{\max}$ at the beginning and at the end of the phase, where $d_{\max}$ denotes the maximal label $d(v)$ over all active vertices v. Note that $d_{\max}$ decreases monotonically during a phase; immediately after the end of the phase, when a RELABEL-operation is executed, $d_{\max}$ increases again.

We claim that the sum of the lengths l_i over all the phases is at most $O(|V|^2)$. To see this, it suffices to show that the increase of $d_{\max}$ during the entire algorithm is at most $O(|V|^2)$. But this is an immediate consequence of Lemma 6.6.8, since the label $d(v)$ increases monotonically for each vertex v and is always bounded by $2|V| - 1$.

The basic idea of the proof is to partition the non-saturating PUSH-operations in a clever way. For this purpose, we call a non-saturating PUSH(u, v)-operation *special*[13] if it is the first PUSH-operation on the edge uv following a RELABEL(u)-operation.

Now consider a non-saturating, nonspecial PUSH-operation PUSH(z, w). We try to construct (in reverse order) a directed path T_w with end vertex w which consists entirely of edges for which the last non-saturating PUSH-operation executed was a nonspecial one. Suppose we have reached a vertex $u \neq w$, and let the last edge constructed for T_w be uv. Thus the last PUSH(u, v) was a non-saturating nonspecial PUSH. Before this PUSH-operation was executed, we had $e(u) > 0$. We want to consider the last PUSH-operation PUSH(y, u) executed before this PUSH(u, v). It is possible that no such PUSH-operation exists;[14] then we simply end the construction of T_w at the vertex u. We also terminate the construction of T_w at u if the last

[12] As the proof of Theorem 6.6.17 is rather technical, the reader might decide to skip it at first reading. However, it does involve a useful method, which we have not seen before.

[13] In the original paper, the term *non-zeroing* is used instead.

[14] Note that this case occurs if and only if the entire flow excess in u comes directly from s, that is, if it was assigned to u during the initialization phase.

PUSH(y, u) was saturating or special. Otherwise we replace u by y and continue in the same manner.

Note that our construction has to terminate provided that T_w is indeed a path, which just amounts to showing that no cycle can occur during the construction. But this is clear, as PUSH-operations may only move flow towards vertices with lower labels; hence no cycle can arise, unless a RELABEL occurred for one of the vertices that we have reached; and this is not possible by our way of construction. We call the sequence of non-saturating PUSH-operations corresponding to such a path T_w a *trajectory* with *originating edge* xy, if xy is the unique edge encountered at the end of the construction of T_w for which either a saturating or a special PUSH had been executed (so that the construction was terminated at y); in the exceptional case mentioned above, we consider the edge su to be the originating edge of T_w. By definition, the originating edge is *not* a part of T_w: the trajectory starts at the head of this edge.

We claim that the whole of the nonspecial non-saturating PUSH-operations can be partitioned into such trajectories. Actually we require a somewhat stronger statement later: two trajectories containing PUSH-operations on edges which are current edges simultaneously (in different adjacency lists) cannot have any vertices in common, with the exception of possible common end vertices. We may assume w.l.o.g. that the two trajectories correspond to paths T_w and $T_{w'}$ for which (at a certain point of the algorithm) both $e(w) > 0$ and $e(w') > 0$ hold. Let xy and $x'y'$ be the originating edges of the two trajectories. Now suppose that u is a common vertex contained in both trajectories, where $u \neq y, y', w, w'$. We may also choose u to be the last such vertex. Let uv and uv' be the edges occurring in T_w and $T_{w'}$, respectively. We may assume that PUSH(u, v) was executed before PUSH(u, v'); note that $v \neq v'$ by our choice of u. Then PUSH(u, v') can only have been executed after some flow excess was moved to u again by some PUSH(z, u)-operation. Then the condition $d(z) = d(u) + 1$ must have been satisfied; this means that there must have been a RELABEL(z)-operation executed before, since the active vertex is always a vertex having a maximal label and u was already active before z. Thus the PUSH(z, u)-operation was a special PUSH and the construction of $T_{w'}$ should have been terminated at the vertex u with the originating edge zu, contradicting the choice of u. Hence any two trajectories are always disjoint, establishing our claim.

Let us call a trajectory *short* if it consists of at most K operations; here K is a parameter whose value we will fix later in an optimal manner. As the originating edge of any trajectory comes from a saturating or a special PUSH-operation or – in the exceptional case – from the initialization of the preflow, the number of short trajectories can be bounded by $O(|V||E|)$ as follows. By Lemma 6.6.10, there are at most $O(|V||E|)$ saturating PUSH-operations. Also, there are at most $O(|V||E|)$ special PUSH-operations, since there are at most $O(|V|)$ RELABEL-operations per vertex by Lemma 6.6.9 and since a special PUSH(u, v) has to be preceded by a RELABEL(u). Hence all the

short trajectories together may contain at most $O(K|V||E|)$ non-saturating PUSH-operations.

Now we have to examine the *long* trajectories, that is, those trajectories which contain more than K operations. Recall that any two trajectories containing PUSH-operations on edges which are current simultaneously cannot contain any common vertices (excepting the end vertices). Hence, at any point during the course of the algorithm, there are at most $|V|/K$ long trajectories which contain a PUSH-operation current at this point. In particular, there are at most $|V|/K$ long trajectories meeting a given phase of the algorithm. By definition, no phase contains a RELABEL-operation, and PUSH(u, v) can only be executed for $d(u) = d(v)+1$. Hence there can be only l_i non-saturating PUSH-operations per trajectory in any given phase of length l_i, as l_i is the difference between the values of $d_{\max}$ at the beginning and at the end of phase i: if PUSH-operations have been executed on a path of length c during phase i, the maximal label must have been decreased by c at least. As we already know that the sum of all lengths l_i is $O(|V|^2)$, all the long trajectories together may contain at most $O(|V|^3/K)$ non-saturating PUSH-operations.

Altogether, the entire algorithm uses at most $O(K|V||E|) + O(|V|^3/K)$ non-saturating PUSH-operations. Now we get the optimal bound on the complexity by *balancing* these two terms, that is, by choosing K in such a way that $K|V||E| = |V|^3/K$. This gives $K = |V||E|^{-q/2}$ and leads to the complexity stated in the assertion. □

Balancing techniques as in the proof above are a useful tool for analyzing the complexity of algorithms. Cheriyan and Maheshwari have also shown that the bound in Theorem 6.6.17 is best possible: there exist families of networks for which Algorithm 6.6.16 indeed needs $O(|V|^2|E|^{1/2})$ steps. From a practical point of view, Algorithm 6.6.16 is one of the best methods known for determining maximal flows; see the empirical studies mentioned at the end of Section 6.4.

Example 6.6.18. Let us apply Algorithm 6.6.16 to the flow network of Figure 6.2; compare Example 6.2.3. Where a choice has to be made, we use alphabetical order, as usual. We summarize several operations in each figure, namely at least one RELABEL-operation together with all PUSH-operations following it; sometimes we even display two or three shorter phases in one figure. We will not draw pairs of antiparallel edges: we include only those edges which carry a nonnegative flow, in accordance with the intuitive interpretation discussed at the beginning of this section; this simplifies the figures.

Moreover, we give the capacities in parentheses only in the first figure (after the initialization phase). The numbers in the subsequent figures always give the values as they are after the last operation executed. So the number written on some edge e is the value $f(e)$ of the current preflow f; here all new values coming from a saturating PUSH-operation are framed, whereas all new values coming from a non-saturating PUSH-operation are circled. Additionally, the vertices v are labelled with the pair $(d(v), e(v))$; that is, we

display the valid label and the flow excess in v. For the vertices s and t, only the (never changing) valid label is given; by definition, these two vertices are never active.

Below each figure, we also list the RELABEL- and PUSH-operations which have occurred and the queue Q containing the active vertices as it looks after all the operations have been executed. Note that the maximal flow constructed by HLFLOW given in Figure 6.31 differs from the maximal flow of Figure 6.12: the edge ct does not carry any flow, and the value of the flow on the edges cf and ft is larger accordingly.

Exercise 6.6.19. Apply Algorithm 6.6.14 to the flow network of Figure 6.2 (with the usual convention about alphabetical order), and compare the number of RELABEL- and PUSH-operations necessary with the corresponding numbers for Algorithm 6.6.16; see Example 6.6.18.

For a discussion of the implementation of various PUSH- and RELABEL-algorithms, see [ChGo95].

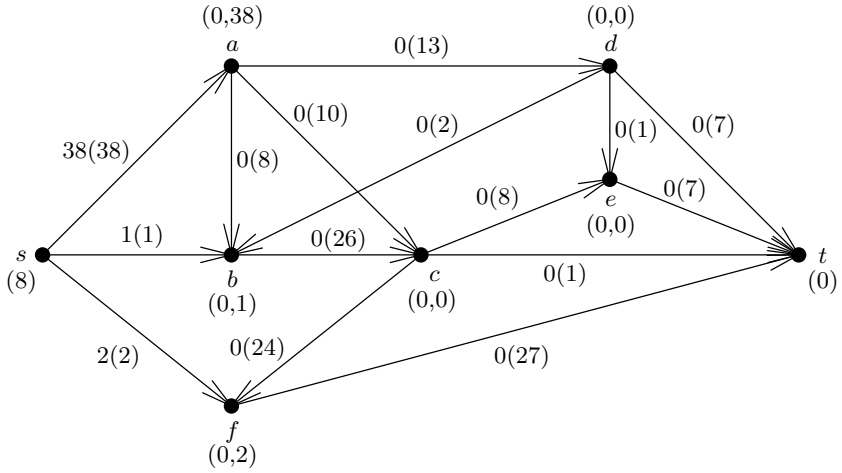


Fig. 6.24. Initialization: $Q = (a, b, f)$

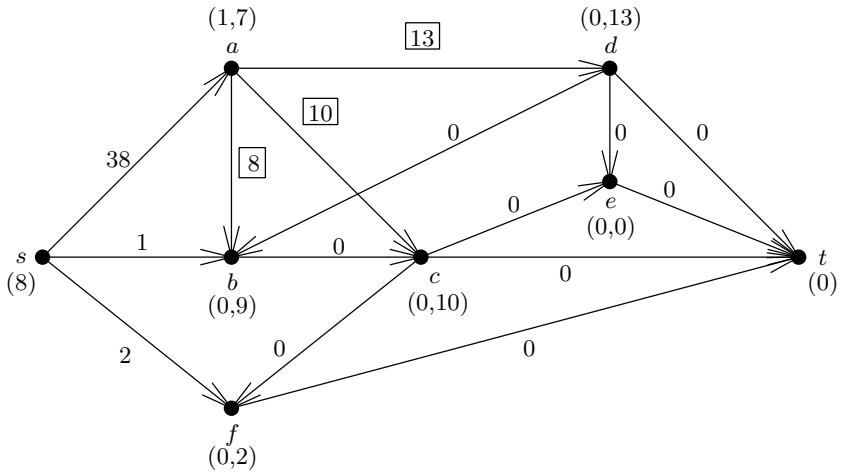


Fig. 6.25. RELABEL(a), $Q = (a, b, c, d, f)$

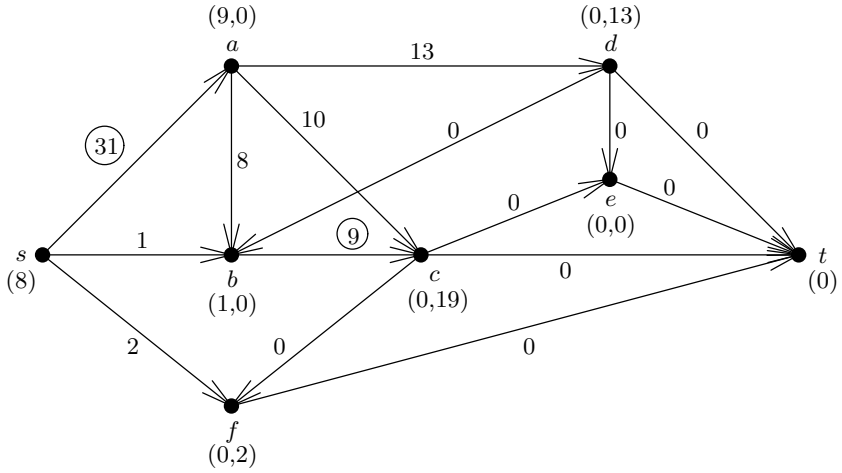


Fig. 6.26. RELABEL(a), PUSH(a, s), RELABEL(b), PUSH(b, c),$Q = (c, d, f)$

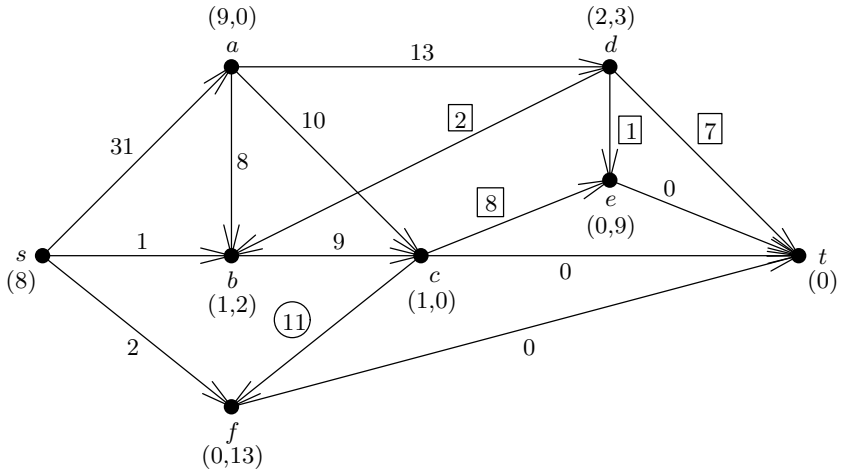


Fig. 6.27. RELABEL(c), PUSH(c, f), RELABEL(d), PUSH(d, b), PUSH(d, e), PUSH(d, t), RELABEL(d), $Q = (d, b, e, f)$

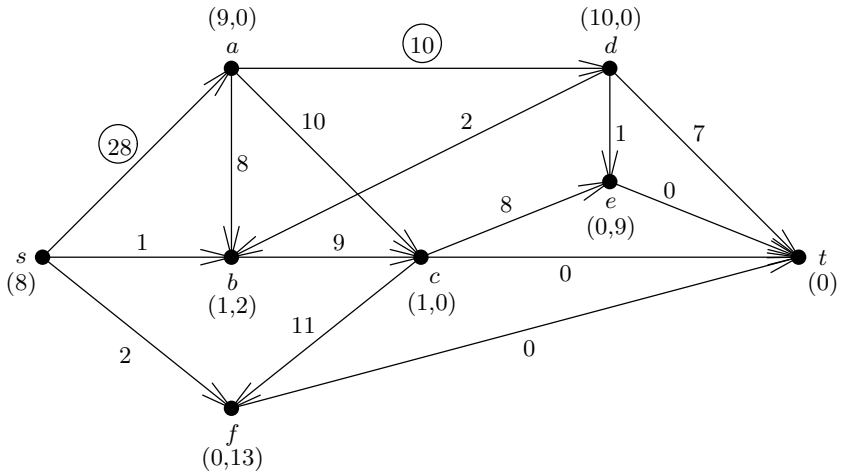


Fig. 6.28. RELABEL(d), PUSH(d, a), PUSH(a, s), $Q = (b, e, f)$

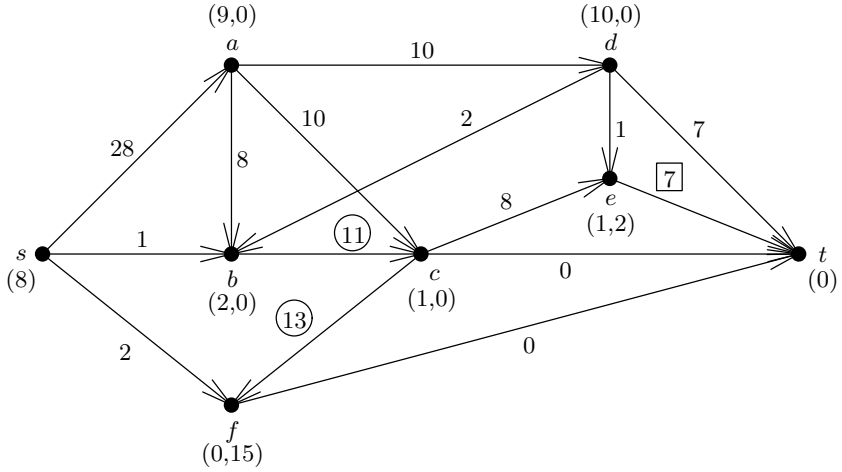


Fig. 6.29. RELABEL(b), PUSH(b, c), PUSH(c, f), RELABEL(e),PUSH(e, f), $Q = (e, f)$

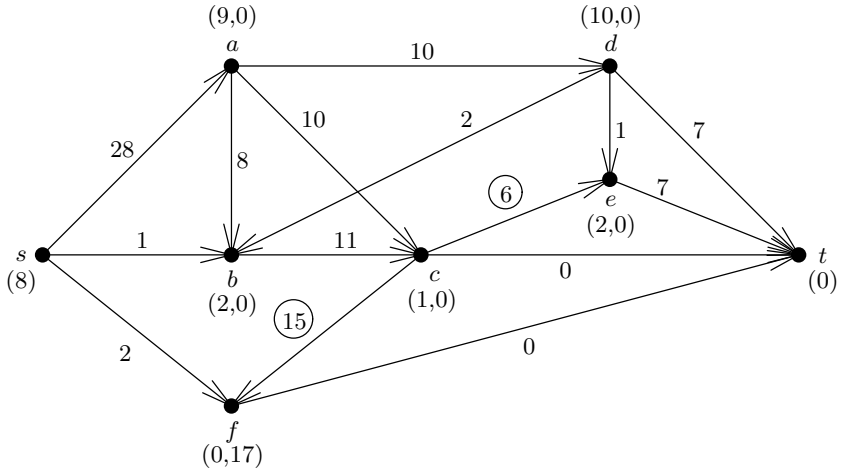


Fig. 6.30. RELABEL(e), PUSH(e, c), PUSH(c, f), $Q = (f)$

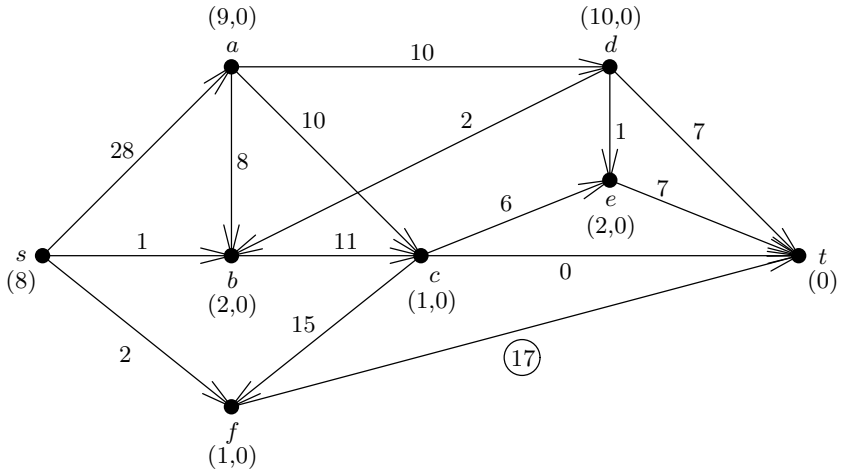


Fig. 6.31. RELABEL(f), PUSH(f, t), $Q = \emptyset$

7

Combinatorial Applications

Everything flows.
HERACLITUS

In this chapter, we use the theorems of Ford and Fulkerson about maximal flows to prove some central results in combinatorics. In particular, transversal theory can be developed from the theory of flows on networks; this approach was first suggested in the book by Ford and Fulkerson [FoFu62] and is also used in the survey [Jun86]. Compared with the usual approach [Mir71b] of taking Philip Hall's marriage theorem [Hal35] – which we will treat in Section 7.3 – as the starting point of transversal theory, this way of proceeding has a distinct advantage: it also yields algorithms for explicit constructions. We shall study disjoint paths in graphs, matchings in bipartite graphs, transversals, the combinatorics of matrices, partitions of directed graphs, partially ordered sets, parallelisms, and the supply and demand theorem.

7.1 Disjoint paths: Menger's theorem

The theorems treated in this section are variations of one of the most widely known results in graph theory, namely Menger's theorem. All these theorems deal with the number of disjoint paths joining two vertices of a graph or a digraph. There are two possible definitions of what *disjoint* means here. Let G be a graph and s and t two vertices of G. Then a set of paths in G with start vertex s and end vertex t is called *edge disjoint* if no two of these paths share an edge, and *vertex disjoint* if no two of the paths have a vertex other than s and t in common. A subset A of E is called an *edge separator* for s and t if each path from s to t contains some edge from A. Similarly, a subset X of $V \setminus \{s, t\}$ is called a *vertex separator* for s and t if each path from s to t meets X in some vertex. If G is a digraph, we use the same terminology but assume the paths in question to be directed. The following theorem – although quite similar to the original theorem of Menger [Men27] – was published much later; see [FoFu56] and [ElFS56]; we shall derive it from the max-flow min-cut theorem.

Theorem 7.1.1 (Menger's theorem, edge version). *Let G be a graph or digraph, and let s and t be two vertices of G. Then the maximal number of edge disjoint paths from s to t is equal to the minimal cardinality of an edge separator for s and t.*

Proof. First let G be a digraph. We may assume that t is accessible from s; otherwise, the assertion is trivial. Let us consider the network N on G where each edge has capacity $c(e) = 1$, and let k denote the maximal number of edge disjoint directed paths from s to t. Obviously, any system of such paths yields a flow f of value k by putting $f(e) = 1$ if e occurs in one of the paths, and $f(e) = 0$ otherwise. Hence the maximal value of a flow on N is some integer $k' \geq k$. The proof of Theorem 6.1.5 shows that we can construct an integral maximal flow by beginning with the zero flow and then using k' augmenting paths of capacity 1. Note that these paths are not necessarily directed, as backward edges might occur. Nevertheless, it is always possible to find k' augmenting paths *without* backward edges: suppose that $e = uv$ is a backward edge occurring in the path W; then there has to exist a path W' which was constructed before W and which contains e as a forward edge. Thus the paths W and W' have the form

$$W = s \xrightarrow{W_1} v \xrightarrow{e} u \xrightarrow{W_2} t$$

and

$$W' = s \xrightarrow{W_1'} u \xrightarrow{e} v \xrightarrow{W_2'} t.$$

Now we may replace the paths W and W' by the paths W_1W_2' and $W_1'W_2$ and thus eliminate the edge e. We may assume that e is the backward edge which occurred first; then W_1, W_2' and W_1' contain forward edges only (whereas W_2 might still contain backward edges). Repeating this construction as often as necessary, we obtain k' augmenting paths which consist of forward edges only, that is, directed paths from s to t in G. Any two augmenting paths consisting of forward edges have to be edge disjoint, as all edges have unit capacity. This implies $k' \leq k$, and hence $k = k'$.

Thus the maximal number of edge disjoint paths from s to t in G is equal to the maximal value of a flow on N and hence, by Theorem 6.1.6, to the capacity of a minimal cut in N. It remains to show that the minimal cardinality of an edge separator for s and t is equal to the capacity of a minimal cut in N. Obviously, any cut (S, T) in N yields an edge separator of cardinality $c(S, T)$:

$$A = \{e \in E\colon e^- \in S, e^+ \in T\}.$$

Conversely, let A be a given minimal edge separator for s and t. Denote the set of those vertices v which are accessible from s by a directed path containing no edges of A by S_A, and put $T_A = V \setminus S_A$. Then (S_A, T_A) is a cut in N. Looking at the definition of the sets S_A and T_A, it is clear that each edge e with $e^- \in S_A$ and $e^+ \in T_A$ has to be contained in A. As A is minimal, A

consists of exactly these edges and is therefore induced by a cut. This proves the theorem for the directed case.

Now let G be a graph. We reduce this case to the directed case by considering the complete orientation $\vec{G}$ of G. Obviously, a system of edge disjoint paths in G induces a corresponding system of edge disjoint directed paths in $\vec{G}$. The converse also holds, provided that the edge disjoint directed paths in $\vec{G}$ do not contain any pair of antiparallel edges. But such pairs of edges can be eliminated, similar to the elimination of backward edges in the first part of the proof. Now let k be the maximal number of edge disjoint directed paths in G and hence also in $\vec{G}$. Then there exists an edge separator $\vec{A}$ in $\vec{G}$ of cardinality k; the corresponding set of edges in G is an edge separator for s and t in G of cardinality $\leq k$. As the minimal cardinality of an edge separator for s and t has to be at least as large as the maximal number of disjoint paths from s to t, we obtain the assertion. □

The proof of Theorem 7.1.1 shows that we may use the algorithm of Dinic to construct a maximal 0-1-flow (of value k, say), and then find k edge disjoint paths from s to t by eliminating backward edges. The algorithm should be modified for this task so that it immediately eliminates a backward edge whenever such an edge occurs. The reader is asked to provide such a modification and convince himself that this does not increase the complexity of the algorithm. In view of Theorems 7.1.1 and 6.5.3, we get the following result.

Corollary 7.1.2. *Let G be a (directed) graph and s and t two vertices of G. Then the maximal number of (directed) edge disjoint paths from s to t (and a system of such paths) can be determined with complexity $O(|V|^{2/3}|E|)$.* □

Exercise 7.1.3. Let N be any flow network. Show that one may construct a maximal flow using augmenting paths which consist of forward edges only; do so for the flow network of Example 6.2.3. Hint: Apply a method similar to that used in the proof of Theorem 7.1.1.

Now we turn to vertex disjoint paths. The analogue of Theorem 7.1.1 is the following well-known result due to Menger [Men27].

Theorem 7.1.4 (Menger's theorem, vertex version). *Let G be a graph or digraph, and let s and t be any two non-adjacent vertices of G. Then the maximal number of vertex disjoint paths from s to t is equal to the minimal cardinality of a vertex separator for s and t.*

Proof. We assume that G is a digraph; the undirected case can be treated in a similar manner. In order to reduce the assertion to Theorem 7.1.1, we define a new digraph G' as follows. Loosely speaking, we split each vertex different from s and t into two parts joined by an edge; this will result in transforming vertex disjoint paths into edge disjoint paths.

Formally, the vertices of G' are s, t, and, for each vertex $v \neq s, t$ of G, two new vertices v' and v''. For every edge sv or vt in G, G' contains the edge sv'

or $v''t$, respectively; and for every edge uv in G, where $u, v \neq s, t$, G' contains the edge $u''v'$. Finally, G' also contains all edges of the form $v'v''$, where v is a vertex of G with $v \neq s, t$. It is clear that vertex disjoint paths in G indeed correspond to edge disjoint paths in G'.

By Theorem 7.1.1, the maximal number of edge disjoint paths from s to t in G' equals the minimal cardinality of an edge separator for s and t, say A. Of course, A might contain edges not of the form $v'v''$, in which case it would not immediately correspond to a vertex separator in G. However, if some edge $u''v'$ occurs in A, we may replace it by $u'u''$ and obtain again a minimal edge separator. Hence we may restrict our considerations to minimal edge separators in G' which only contain edges of the form $v'v''$ and therefore correspond to vertex separators in G. □

Corollary 7.1.5. *Let G be a graph or digraph, and let s and t be any two non-adjacent vertices of G. Then the maximal number of vertex disjoint paths from s to t – and a system of such paths – can be determined with complexity $O(|V|^{1/2}|E|)$.*

Proof. We may assume w.l.o.g. that all vertices of G are accessible from s. Then the digraph G' constructed in the proof of Theorem 7.1.4 has $O(|V|)$ vertices and $O(|E|)$ edges. The assertion follows in the same way as Corollary 7.1.2 did, taking into account that the network defined on G' (with capacity 1 for all edges) satisfies the condition of Theorem 6.5.4. □

The existence of disjoint paths plays an important role for questions of *network reliability*: if there are k vertex disjoint paths from s to t, the connection between s and t can still be maintained even if $k-1$ vertices fail, and similarly for edges. Such considerations are important for computer networks, for example.[1] This suggests measuring the strength of connectivity of a connected graph by the number of vertex disjoint paths (or edge disjoint paths) between any two given vertices. Menger's theorem leads to the following definition.

Definition 7.1.6. The *connectivity* $\kappa(G)$ of a graph $G = (V, E)$ is defined as follows. If G is a complete graph K_n, then $\kappa(G) = n - 1$; otherwise

$$\kappa(G) \;=\; \min\{|T|\colon T \subset V \text{ and } G \setminus T \text{ is not connected}\}\,.$$

G is called *k-connected* if $\kappa(G) \geq k$.

We will consider questions of connectivity in Chapter 8 in detail; now we just pose three exercises.

Exercise 7.1.7 (Whitney's theorem). Show that a graph G is k-connected if and only if any two vertices of G are connected by at least k vertex disjoint paths [Whi32a]. (Hint: Note that Menger's theorem only applies to non-adjacent vertices s and t.)

[1] For more on network reliability, we recommend [Col87].

Exercise 7.1.8. Use Exercise 1.5.14 to show that a planar graph can be at most 5-connected. Moreover, find a 4-connected planar graph on six vertices; also, show that a 5-connected planar graph has at least 12 vertices, and give an example on 12 vertices.

Exercise 7.1.9. Let S and T be two disjoint subsets of the vertex set V of a graph $G = (V, E)$. Show that the minimal cardinality of a vertex separator X for S and T (that is, every path from some vertex in S to some vertex in T has to contain some vertex in X) is equal to the maximal number of paths from S to T such that no two of these paths have any vertex in common (not even one of the end vertices!).

Exercise 7.1.10. We have proved the vertex version of Menger's theorem (Theorem 7.1.4) by reducing it to the edge version (Theorem 7.1.1). However, it is also possible to go the opposite way and deduce Theorem 7.1.1 from Theorem 7.1.4. Do so in the undirected case. Hint: The required transformation is similar to the construction of the line graph in Section 1.3.

7.2 Matchings: König's theorem

Recall that a *matching* in a graph G is a set M of edges no two of which have a vertex in common. In this section, we consider matchings in bipartite graphs only; the general case will be dealt with in Chapter 13. The following result was already proved in Example 6.5.5.

Theorem 7.2.1. *Let G be a bipartite graph. Then a matching of maximal cardinality in G can be determined with complexity $O(|V|^{5/2})$.* □

It is common usage to call a matching of maximal cardinality a *maximal matching*. This is really quite misleading, as the term *maximal* suggests that such a matching cannot be extended to a larger matching; however, an *unextendable* matching does not necessarily have maximal cardinality, as the example in Figure 7.1 shows. Still, we will often accept such ambiguity, as this unfortunate terminology is firmly established.

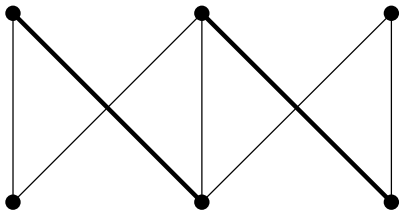


Fig. 7.1. A matching which cannot be extended

Exercise 7.2.2. Let G be an arbitrary (not necessarily bipartite) graph, and denote the maximal cardinality of a matching in G by k. Find a lower bound for the number of edges of an unextendable matching.

Note that Theorem 7.2.1 is a special case of Corollary 7.1.5. To see this, let $G = (S \dot\cup T, E)$ be the given bipartite graph. We define a new graph H which has, in addition to the vertices of G, two new vertices s and t and whose edges are the edges of G plus all edges sx for $x \in S$ and all edges yt for $y \in T$. Obviously, the edges of a matching M in G correspond to vertex disjoint paths from s to t in H: associate the path $s \longrightarrow x \longrightarrow y \longrightarrow t$ with the edge xy in G, where $x \in S$ and $y \in T$. Of course, for determining a maximal matching in practice, we should not use Corollary 7.1.5 (and the graphs used there) but work with H itself as described in Example 6.5.5.

Let us apply Theorem 7.1.4 to the graph H just defined and interpret this result within G. As noted above, vertex disjoint paths in H correspond to matchings in G. Also, a vertex separator for s and t in H is a set X of vertices in G such that each edge of G has at least one of its end vertices in X; that is, X is a vertex cover for G. It is usual to denote the maximal cardinality of a matching by $\alpha'(G)$, and the minimal cardinality of a vertex cover by $\beta(G)$. Using this notation, Theorem 7.1.4 immediately implies the following major result [Koe31, Ege31]: $\alpha'(G) = \beta(G)$.[2] We shall provide a second proof taken from [Riz00] which does not rely on Menger's theorem and, hence, on the theory of network flows; this also provides a nice illustration for a further important method in discrete mathematics: proof by *minimal counterexample*.

Theorem 7.2.3 (König's theorem). *Let G be bipartite graph. Then the maximal cardinality of a matching in G equals the minimal cardinality of a vertex cover: $\alpha'(G) = \beta(G)$.*

Proof. By definition, no vertex can be incident with more than one edge in a given matching. Hence one direction of the assertion is obvious: $\alpha'(G) \leq \beta(G)$. Thus we only need to prove the reverse inequality.

Now let us assume to the contrary that $\alpha'(G) < \beta(G)$. Among all bipartite graphs violating the theorem, we choose $G = (S \dot\cup T, E)$ as a minimal counterexample: G has the smallest possible number of vertices, say n; and among all counterexamples on n vertices, G also has the minimal number of edges. Then G is connected. Also, G cannot be a cycle or a path, since bipartite graphs of this type clearly satisfy the theorem. Hence we may choose a vertex u with $\deg u \geq 3$. Let v be adjacent to u, and consider the graph $G \setminus v$. Since G was chosen as a minimal counterexample, $G \setminus v$ satisfies the theorem: $\alpha'(G \setminus v) = \beta(G \setminus v)$.

Now assume $\alpha'(G \setminus v) < \alpha'(G)$. Then we may adjoin v to a vertex cover W of $G \setminus v$ with cardinality $\alpha'(G \setminus v)$ to obtain a vertex cover for G. This

[2] Quite often, this result is stated in the language of matrices instead; see Theorem 7.4.1 below.

implies $\beta(G) \leq \alpha'(G \setminus v) + 1 \leq \alpha'(G)$, and G satisfies the theorem after all, a contradiction.

Hence we must have $\alpha'(G \setminus v) = \alpha'(G)$. Then there exists a maximal matching M of G for which no edge in M is incident with v. In view of $\deg u \geq 3$, we may choose an edge $e \notin M$ which is incident with u, but not with v. Because of the minimality of G, the subgraph $G \setminus e$ satisfies the theorem, and we obtain $\alpha'(G) = \alpha'(G \setminus e) = \beta(G \setminus e)$. Let W' be a vertex cover of $G \setminus e$ with cardinality $\alpha'(G)$. As no edge in M is incident with v, we must have $v \notin W'$. Hence W' has to contain the other end vertex u of the edge $uv \neq e$ and is therefore actually a vertex cover for G. Again, G satisfies the theorem after all; this final contradiction establishes the theorem. □

Exercise 7.2.4. Use the problem transformation described in Example 6.5.5 to derive Theorem 7.2.3 directly from Theorem 6.1.6, without the detour via Menger's theorem made before.

Hint: Consider a maximal flow and a minimal cut determined by the labelling algorithm, and study the interaction of the cut with the associated maximal matching.

Obviously, the maximal cardinality of a matching in a bipartite graph $G = (S \dot\cup T, E)$ is bounded by $\min\{|S|, |T|\}$. A matching of this cardinality is called a *complete matching*. The following theorem due to Philip Hall [Hal35] characterizes the bipartite graphs which admit a complete matching.[3]

Theorem 7.2.5. *Let $G = (S \dot\cup T, E)$ be a bipartite graph with $|S| \geq |T|$. For $J \subset T$, let $\Gamma(J)$ denote the set of all those vertices in S which are adjacent to some vertex in J. Then G admits a complete matching if and only if the following condition is satisfied:*

(H) $\quad |\Gamma(J)| \geq |J| \quad$ *for all $J \subset T$.*

Proof. To see that condition (H) is necessary, let M be a complete matching of G and J any subset of T. Denote the set of edges contained in M which are incident with a vertex in J by $E(J)$. Then the end vertices of the edges in $E(J)$ which are contained in S form a subset of cardinality $|J|$ of $\Gamma(J)$. Conversely, suppose that condition (H) is satisfied and that the maximal cardinality of a matching in G is less than $|T|$. Then Theorem 7.2.3 yields the existence of a vertex cover $X = S' \dot\cup T'$ with $S' \subset S$, $T' \subset T$, and $|S'| + |T'| < |T|$. But then the end vertices u of those edges uv for which v is one of the $|T| - |T'|$ vertices in $T \setminus T'$ are all contained in S', so that

$$|\Gamma(T \setminus T')| \;\leq\; |S'| \;<\; |T| - |T'| \;=\; |T \setminus T'|,$$

a contradiction. □

For $|S| = |T|$, a complete matching is precisely a 1-factor of G; in this case, we also speak of a *perfect matching*. An important consequence of Theorem

[3] This theorem is likewise often stated in different language; see Theorem 7.3.1.

7.2.5 is the following sufficient condition for the existence of perfect matchings. We need a further definition: a *regular bipartite graph* is a bipartite graph $G = (S \dot\cup T, E)$ for which all vertices have the same degree $\neq 0$. Note that this implies $|S| = |T|$.

Corollary 7.2.6. *Let $G = (S \dot\cup T, E)$ be a regular bipartite graph. Then G has a perfect matching.*

Proof. By Theorem 7.2.5, it is sufficient to show that G satisfies condition (H). Let r be the degree of the vertices of G. If J is a k-subset of T, there are exactly kr edges of the form st with $t \in J$ and $s \in S$. As each vertex in S is incident with exactly r edges, these kr edges have to be incident with at least k distinct vertices in S. □

Corollary 7.2.7. *A bipartite graph G has a 1-factorization if and only if it is regular.*

Proof. Obviously, the regularity of G is necessary. Using induction, Corollary 7.2.6 shows that this condition is also sufficient. □

Exercise 7.2.8. Show that an r-regular non-bipartite graph does not necessarily admit a 1-factorization, even if it has an even number of vertices. Hint: Consider the Petersen graph.

The following – somewhat surprising – application of Corollary 7.2.7 is due to Petersen [Pet91].

Theorem 7.2.9. *Every $2k$-regular graph (where $k \neq 0$) has a 2-factorization.*

Proof. Let G be a $2k$-regular graph and assume w.l.o.g. that G is connected. By Theorem 1.3.1, G contains an Euler tour C. Let H be an orientation of G such that C is a directed Euler tour for H. Now we define a regular bipartite graph G' as follows. For each vertex v of H, let G' have two vertices v' and v''; and for every edge uv of H, let G' contain an edge $u'v''$. Then G' is k-regular, and hence G' has a 1-factorization, by Corollary 7.2.7. It is easy to see that each 1-factor of G' corresponds to a 2-factor of G, so that we get a 2-factorization for G. □

We close this section with some exercises concerning factorizations.

Exercise 7.2.10. Let G be a graph on $3n$ vertices. A 2-factor of G is called a *triangle factor* or a *$\triangle$-factor* if it is the disjoint union of n cycles of length 3. Show that it is possible to decompose the graph K_{6n} into one $\triangle$-factor and $6n - 3$ 1-factors. Hint: View the vertex set as the union of three sets R, S, T of cardinality $2n$ each, and consider regular bipartite graphs on all pairs of these sets. Furthermore, use Exercise 1.1.2.

△-factors are used in finite geometry; for example, Exercise 7.2.10 is used in [JuLe87] for constructing certain *linear spaces*. The general problem of decomposing K_{6n} into c △-factors and d 1-factors was studied by Rees [Ree87]. It is always possible to decompose K_{6n} into a 1-factor and $3n-1$ △-factors yielding a so-called *near-Kirkman triple system*; see [BaWi77] and [HuMR82].

The most popular problem in this context is the case of *Kirkman triple systems*, which are decompositions of K_{6n+3} into △-factors. The name comes from a famous problem in recreational mathematics, namely *Kirkman's school girl problem*, which was posed in [Kir50] as follows:

> *Fifteen young ladies in a school walk out three abreast for seven days in succession; it is required to arrange them daily, so that no two will walk twice abreast.*

If we represent the school girls by 15 vertices and join two of them by an edge if they walk abreast, then a daily arrangement corresponds to a △-factor of K_{15}, and the seven △-factors for the seven days form a decomposition into △-factors. A solution of this problem is given in Figure 7.2, where only one △-factor is drawn. The other △-factors in the decomposition are obtained by rotating the given factor around the vertex ∞ so that the set $\{p_0, \ldots, p_6\}$ is left invariant; there are seven ways to do so, including the identity mapping. The general problem of decomposing the graph K_{6n+3} into △-factors was only solved 120 years later by Ray-Chaudhuri and Wilson [RaWi71]; see also [BeJL99], §IX.6.

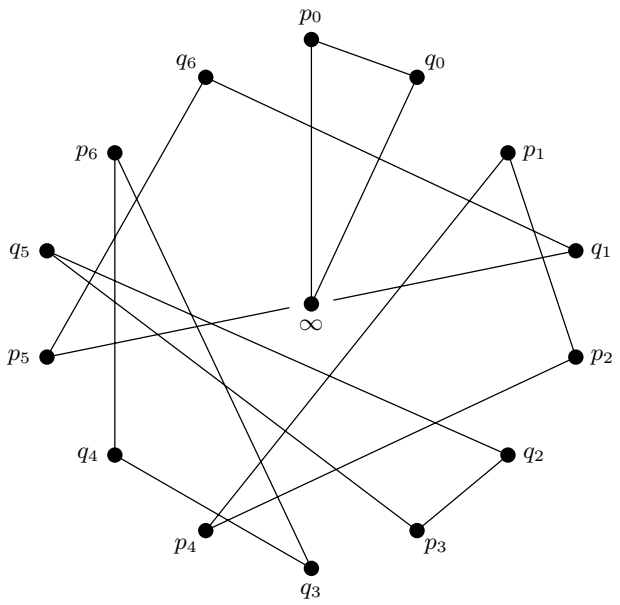


Fig. 7.2. A solution of Kirkman's school girl problem

Exercise 7.2.11. Decompose the graph K_9 into $\triangle$-factors. Hint: There is no *cyclic* decomposition as in Figure 7.2.

Exercise 7.2.12. Decompose the graph K_{6n-2} into 3-factors. Hint: Use Theorem 7.2.9.

Readers interested in seeing more results on 1-factorizations and graph decompositions in general should consult the monographs [Bos90] and [Wal97].

7.3 Partial transversals: The marriage theorem

This section presents the basic theory of transversals. We begin with some definitions. Let $\mathbf{A} = (A_1, \ldots, A_n)$ be a family of subsets of a (finite) set S. Then any family $(a_1, \ldots, a_n)$ with $a_j \in A_j$ for $j = 1, \ldots, n$ is called a *system of representatives* for $\mathbf{A}$. If, in addition, $a_i \neq a_j$ holds whenever $i \neq j$, then $(a_1, \ldots, a_n)$ is called a *system of distinct representatives* (SDR) for $\mathbf{A}$, and the underlying set $\{a_1, \ldots, a_n\}$ is called a *transversal* of $\mathbf{A}$.[4]

Let us construct a bipartite graph G with vertex set $S \mathbin{\dot\cup} T$, where $T = \{1, \ldots, n\}$, which has an edge st whenever $s \in A_t$. Then, for $J \subset T$, the set $\Gamma(J)$ defined in Theorem 7.2.5 is the union of all sets A_t with $t \in J$; and a perfect matching of G is the same as an SDR for $\mathbf{A}$. Therefore Theorem 7.2.5 translates into the following result.

Theorem 7.3.1 (marriage theorem). *Let $\mathbf{A} = (A_1, \ldots, A_n)$ be a family of subsets of a finite set S. Then $\mathbf{A}$ has a transversal if and only if the following condition is satisfied:*

$$\text{(H}') \qquad \left|\bigcup_{j\in J} A_j\right| \geq |J| \qquad \text{for all } J \subset \{1, \ldots, n\}. \qquad \square$$

This theorem was proved by Philip Hall [Hal35] in terms of set families; as explained above, this is equivalent to using the setting of bipartite graphs. The name *marriage theorem* is due to the following interpretation of the theorem. Let S be a set of girls, and view the index set T as a set of boys; the set A_t is the set of girls which boy t would be willing to marry. Then the marriage theorem gives a necessary and sufficient condition for the existence of an arrangement of marriages so that each boy marries some girl of his choice. Of course, the roles of boys and girls may be exchanged. For more symmetry, it is also possible to assume $|S| = |T|$ and to put only those girls into A_t who are actually prepared to accept a proposal from boy t. Then the marriage theorem gives us a criterion if all boys and girls may get a partner of their

[4] For an intuitive interpretation, we might think of the A_i as certain groups of people who each send a representative a_i into a committee. Then the SDR property means that no committee member is allowed to represent more than one group, and the transversal $\{a_1, \ldots, a_n\}$ just is the committee. Another interpretation will be given below, after Theorem 7.3.1.

choice. Thus condition (H′) can be put into everyday language as follows: if nobody is too choosy, everybody can find someone!

The marriage theorem is often considered to be the root of transversal theory, which then appears as a sequence of specializations and applications of this theorem. In particular, the theorems of König, Menger, and Ford and Fulkerson can all be derived from the marriage theorem. The book [Mir71b] uses this approach; let us give two exercises in this direction.

Exercise 7.3.2. Give a direct proof for Theorem 7.3.1 using induction on n. Hint: Use a case distinction depending on the existence of a *critical subfamily* of $\mathbf{A}$, that is, a subfamily $(A_j)_{j\in J}$ with $|\bigcup_{j\in J} A_j| = |J|$.

Exercise 7.3.3. Derive Theorem 7.2.3 from Theorem 7.3.1.

In the remainder of this section, we use the marriage theorem to prove a small selection of further results from transversal theory. We need some definitions first. An SDR for a subfamily $(A_j)_{j\in J}$ of $\mathbf{A} = (A_1, \dots, A_n)$ is said to be a *partial SDR* for $\mathbf{A}$, and the underlying set $\{a_j : j \in J\}$ is called a *partial transversal*. The marriage theorem only distinguishes between families of sets having a transversal and those without transversals. A finer measure for the representability of a family of sets is is the *transversal index* $t(\mathbf{A})$; that is, the maximal cardinality of a partial transversal of $\mathbf{A}$. The *deficiency* of a partial transversal of cardinality k is the number $n - k$; hence the transversal index equals n minus the minimal deficiency of a partial transversal. The following condition of [Ore55] for the existence of a partial transversal with a given deficiency follows easily from the marriage theorem.

Theorem 7.3.4 (deficiency version of the marriage theorem). *Let $\mathbf{A} = (A_1, \dots, A_n)$ be a family of subsets of a finite set S. Then $\mathbf{A}$ has a partial transversal of cardinality k (that is, with deficiency $d = n - k$) if and only if the following condition holds:*

$$\Big|\bigcup_{j\in J} A_j\Big| \;\geq\; |J| + k - n \qquad \text{for all } J \subset \{1, \dots, n\}. \tag{7.1}$$

Proof. Let D be an arbitrary d-set disjoint from S, and define a family $\mathbf{A}' = (A'_1, \dots, A'_n)$ of subsets of $S \cup D$ by putting $A'_i = A_i \cup D$. By Theorem 7.3.1, $\mathbf{A}'$ has a transversal if and only if it satisfies condition (H); that is, if and only if (7.1) holds for $\mathbf{A}$. Now every transversal T of $\mathbf{A}'$ yields a partial transversal for $\mathbf{A}$ of cardinality at least k, namely $T \setminus D$. Conversely, each partial transversal of cardinality k of $\mathbf{A}$ can be extended to a transversal of $\mathbf{A}'$ by adding the elements of D. □

Corollary 7.3.5. *The minimal deficiency of a partial transversal of $\mathbf{A}$ is*

$$d(\mathbf{A}) \;=\; \max\Big\{|J| - \big|\bigcup_{j\in J} A_j\big| \,:\, J \subset \{1, \dots, n\}\Big\},$$

and the transversal index is $t(\mathbf{A}) = n - d(\mathbf{A})$. □

Exercise 7.3.6. Translate Corollary 7.3.5 into the language of bipartite graphs.

Theorem 7.3.7. *Let* $\mathbf{A} = (A_1, \ldots, A_n)$ *be a family of subsets of a finite set* S*. Then a subset* X *of* S *is a partial transversal of* $\mathbf{A}$ *if and only if the following condition holds:*

$$\Big|\bigcup_{j\in J} A_j \cap X\Big| \;\geq\; |J| + |X| - n \quad \textit{for all } J \subset \{1, \ldots, n\}. \tag{7.2}$$

Proof. Obviously, X is a partial transversal of $\mathbf{A}$ if and only if it is a partial transversal of $\mathbf{A}' = (A_i \cap X)_{i=1,\ldots,n}$; that is, if and only if $\mathbf{A}'$ has a partial transversal of cardinality $|X|$. this observation reduces the assertion to Theorem 7.3.4. □

The partial transversals characterized in the preceding theorem are the independent sets of a matroid, a result due to Edmonds and Fulkerson [EdFu65]:

Theorem 7.3.8. *Let* $\mathbf{A} = (A_1, \ldots, A_n)$ *be a family of subsets of a finite set* S*, and let* $\mathbf{S}$ *be the set of partial transversals of* $\mathbf{A}$*. Then* $(S, \mathbf{S})$ *is a matroid.*

Proof. Consider the bipartite graph G corresponding to $\mathbf{A}$ as explained at the beginning of this section. Then the partial transversals of $\mathbf{A}$ are precisely the subsets of the form $\{e^- : e \in M\}$ of S, where M is a matching of G. Hence the assertion reduces to Exercise 6.5.7. □

The matroids described in Theorem 7.3.8 are called *transversal matroids*. Theorems 7.3.8 and 5.2.6 together imply the following result; a constructive proof for this result is given in the solution to Exercise 6.5.7.

Corollary 7.3.9. *Let* $\mathbf{A}$ *be a family of subsets of a finite set* S*, and assume that* $\mathbf{A}$ *has a transversal. Then every partial transversal of* $\mathbf{A}$ *can be extended to a transversal.* □

Corollary 7.3.9 is generally attributed to Hoffmann and Kuhn [HoKu56]. Marshall Hall [Hal56] should also be mentioned in this context; he gave the first algorithm for determining an SDR, and this algorithm yields a constructive proof for Corollary 7.3.9. We note that the solution to Exercise 6.5.7 given in the appendix yields a considerably simpler proof: Hall's algorithm is much harder to understand than the determination of a maximal partial SDR – that is, more precisely, of a maximal matching in the corresponding bipartite graph – using network flows. Moreover, Hall's algorithm does not have polynomial complexity.

Edmonds and Fulkerson also proved a more general version of Theorem 7.3.8 which uses matchings in arbitrary graphs for constructing matroids; we will present this theorem in Section 13.5. The special case above suffices to solve the following exercise:

Exercise 7.3.10. Let $E = A_1 \dot\cup \ldots \dot\cup A_k$ be a partition of a finite set E, and let $d_1, \ldots, d_k$ be positive integers. Then $(E, \mathbf{S})$ is a matroid, where

$$\mathbf{S} \;=\; \{X \subset E\colon |X \cap A_i| \leq d_i \text{ for } i = 1, \ldots, k\}\,.$$

Matroids of this type are called *partition matroids*. If we choose E as the edge set of a digraph G, A_i as the set of all edges with end vertex i, and $d_i = 1$ $(i = 1, \ldots, |V|)$, we get the head-partition matroid of Theorem 5.1.3.

The following strengthening of Corollary 7.3.9 is due to Mendelsohn and Dulmage [MeDu58].

Theorem 7.3.11. *Let $\mathbf{A} = (A_1, \ldots, A_n)$ be a family of subsets of a finite set S. In addition, let $\mathbf{A}'$ be a subfamily of $\mathbf{A}$ and S' a subset of S. Then the following statements are equivalent:*

(1) *$\mathbf{A}'$ has a transversal, and S' is a partial transversal of $\mathbf{A}$.*

(2) *There exist a subset S'' of S containing S' and a subfamily $\mathbf{A}''$ of $\mathbf{A}$ containing $\mathbf{A}'$ for which S'' is a transversal of $\mathbf{A}''$.*

While it is possible to give a direct proof of Theorem 7.3.11 using the methods of transversal theory, it will be simpler and more intuitive to translate the result in question into the language of bipartite graphs; moreover, the symmetry of this result – between subsets of S and subfamilies of $\mathbf{A}$ – is much more transparent in graph theoretic terms. To do so, we need the following definition. Let M be a matching in a graph $G = (V, E)$. We say that M *covers* a subset X of V if each vertex in X is incident with some edge of M. We shall now prove the following result equivalent to (the nontrivial direction of) Theorem 7.3.11.

Theorem 7.3.12. *Let $G = (S \dot\cup T, E)$ be a bipartite graph, S' a subset of S, and T' a subset of T. If there exist matchings M_S and M_T in G covering S' and T', respectively, then there also exists a matching covering $S' \cup T'$.*

Proof. Consider the bipartite subgraph G' of G determined by the edges in $M_S \cup M_T$. Note that the vertex set V' of G' consist precisely of those vertices of G which are covered by M_S or M_T (so that $W := S' \cup T' \subseteq V'$), and that each vertex of G' has degree 1 or 2. Moreover, a vertex v with degree 2 has to be incident with precisely one edge of M_S and one edge of M_T. Therefore the connected components of G' are either cycles of even length, where edges of M_S and M_T alternate, or paths formed by such alternating sequences of edges. We will use these observations to define a matching M of G' – and hence of G – covering W.

Thus consider any connected component C of G'. If C is an (alternating) cycle, we put all edges in $C \cap M_S$ into M; this will cover all vertices in C.[5]

[5] Of course, we might as well choose the edges in $C \cap M_T$.

If C is an (alternating) path of odd length, the first and the last edge of C have to belong to the same matching, say to M_S; again, we put all edges in $C \cap M_S$ into M and cover all vertices in C. Finally, let C be an (alternating) path of even length. Note that the two end vertices of C have to belong to the same part of the bipartition of G, say to S. As C is alternating, one of these two vertices is not covered by M_S (and hence is not contained in S'), since exactly one of the two end edges of C belongs to M_S. Again, we may select all edges in $C \cap M_S$; while this leaves one of the two end vertices exposed, it does cover all the vertices of C contained in W. □

Theorem 7.3.11 now follows by applying Theorem 7.3.12 to the bipartite graph associated with the family $\mathbf{A} = (A_1, \ldots, A_n)$. Nevertheless, a direct proof is a useful exercise:

Exercise 7.3.13. Give a direct proof of Theorem 7.3.11 using the methods of transversal theory. Hint: Consider the auxiliary family $\mathbf{B}$ consisting of the sets $A_1, \ldots, A_k, A_{k+1} \cup D, \ldots, A_n \cup D$ and m times the set $(S \setminus S') \cup D$, where D is an arbitrary set of cardinality n which is disjoint to S.

Exercise 7.3.14. Let $\mathbf{A} = (A_t)_{t \in T}$ be a finite family of subsets of a finite set S. Show that $\mathbf{A}$ induces a matroid on T as well.

The following result of [Hal35] is a further application of the marriage theorem. It gives a criterion for the existence of a common system of representatives for two families of sets.

Theorem 7.3.15. *Let $\mathbf{A} = (A_1, \ldots, A_n)$ and $\mathbf{B} = (B_1, \ldots, B_n)$ be two families of subsets of a finite set S. Then $\mathbf{A}$ and $\mathbf{B}$ have a common system of representatives if and only if the following condition holds:*

$$\left|\left\{i \colon B_i \cap \Big(\bigcup_{j \in J} A_j\Big) \neq \emptyset\right\}\right| \geq |J| \quad \text{for all } J \subset \{1, \ldots, n\}. \tag{7.3}$$

Proof. $\mathbf{A}$ and $\mathbf{B}$ have a common system of representatives if and only if there is a permutation π of $\{1, \ldots, n\}$ such that $A_i \cap B_{\pi(i)} \neq \emptyset$ for $i = 1, \ldots, n$. Define the family $\mathbf{C} = (C_1, \ldots, C_n)$ by $C_j := \{i : A_j \cap B_i \neq \emptyset\}$; then the condition above reduces to the existence of a transversal of $\mathbf{C}$. It is easily seen that condition (H′) for $\mathbf{C}$ is equivalent to (7.3), and thus the assertion follows from Theorem 7.3.1. □

The following two results of [vdW27] and [Mil10], respectively, are immediate consequences of Theorem 7.3.15.

Corollary 7.3.16. *Let $M = A_1 \,\dot\cup\, \ldots \,\dot\cup\, A_n = B_1 \,\dot\cup\, \ldots \,\dot\cup\, B_n$ be two partitions of a finite set M into subsets of cardinality k. Then $(A_1, \ldots, A_n)$ and $(B_1, \ldots, B_n)$ have a common transversal.* □

Corollary 7.3.17. *Let H be any subgroup of a finite group G. Then the families of right and left cosets of H in G necessarily have a common system of representatives.* □

We have proved Theorems 7.3.4, 7.3.7, 7.3.11, and 7.3.15 by applying the marriage theorem to a suitable auxiliary family of sets. Thus, in some sense, the marriage theorem is a *self-strengthening* result, as pointed out by Mirsky [Mir69a]. A further result which can be proved in the same manner is left to the reader as an exercise; see [HaVa50].

Exercise 7.3.18 (harem theorem). Let $\mathbf{A} = (A_1, \ldots, A_n)$ be a family of subsets of a finite set S, and $(p_1, \ldots, p_n)$ a family of positive integers. Show that a family of pairwise disjoint sets $(X_1, \ldots, X_n)$ with $X_i \subset A_i$ and $|X_i| = p_i$ for $i = 1, \ldots, n$ exists if and only if the following condition holds:

$$\Big|\bigcup_{i\in J} A_i\Big| \;\geq\; \sum_{i\in J} p_i \quad \text{for all } J \subset \{1, \ldots, n\}.$$

We close this section with some remarks. Network flow theory can be used to prove many more results about (partial) transversals and systems of representatives; we refer to [FoFu58b, FoFu62]. In particular, it is possible to derive a criterion for when two families of sets have a common transversal. However, this result follows more easily from a generalization of the marriage theorem to matroids due to Rado [Rad42], who gave a criterion for the existence of transversals which are independent in the matroid. It turns out that the theory of matroids is the natural structural setting for transversal theory; we refer to the books [Mir71b, Wel76], to the survey [Mir69b], and to [MiPe67].

7.4 Combinatorics of matrices

This section treats some combinatorial theorems concerning matrices. We begin by translating Theorem 7.2.3 into the language of matrices. Let $A = (a_{ij})_{i=1,\ldots,m;\, j=1,\ldots,n}$ be a matrix where a certain subset of the *cells* (i, j) is marked as *admissible* – usually, the cells (i, j) with $a_{ij} \neq 0$. A set C of cells is called *independent* if no two cells of C lie in the same row or column of A. The *term rank* or *scatter number* $\rho(A)$ is the maximal cardinality of an independent set of admissible cells of A. Corresponding to A, we construct a bipartite graph G with vertex set $S \,\dot\cup\, T$, where $S = \{1, \ldots, m\}$ and $T = \{1', \ldots, n'\}$, and where G contains an edge st' if and only if the cell (s, t) is admissible. Then the matchings of G correspond to the independent sets of admissible cells of A; moreover, vertex covers of G correspond to those sets of rows and columns which contain all the admissible cells. Hence Theorem 7.2.3 translates into the following result.

Theorem 7.4.1. *The term rank $\rho(A)$ of a matrix A is equal to the minimal number of rows and columns of A which contain all the admissible cells of A.* □

From now on, we restrict our attention to square matrices. We want to derive a criterion of Frobenius [Fro12] which tells us when all terms in the sum representation of the determinant of a matrix are equal to 0. Again, we need some definitions. If A is an $n \times n$ matrix, any set of n independent cells is called a *diagonal*. A diagonal is said to be a *non-zero diagonal* or a *positive diagonal* if each of its cells has entry $\neq 0$ or > 0, respectively. The *width* of an $r \times s$ matrix is $r + s$. Now we mark the cells having entry $\neq 0$ as admissible and define a bipartite graph G corresponding to A, as before. Then a non-zero diagonal of A corresponds to a perfect matching of G. We get the following result equivalent to Theorem 7.2.5.

Lemma 7.4.2. *Let A be a square matrix of order n. Then A has a non-zero diagonal if and only if the non-zero entries in a set of k columns of A always belong to at least k different rows.* □

Theorem 7.4.3. *Let A be an $n \times n$ matrix. Then each diagonal of A contains at least one entry 0 if and only if A has a zero submatrix of width $n + 1$.*

Proof. By Lemma 7.4.2, every diagonal of A contains an entry 0 if and only if there are k columns of A which have all their non-zero entries in $r < k$ rows. Then these k columns have entry 0 in the remaining $n - r > n - k$ rows, and we obtain a zero submatrix of width $n - r + k \geq n + 1$. □

Note that the diagonals of A correspond precisely to the terms in the sum representation of the determinant of A, so that Theorem 7.4.3 gives the desired criterion. Next, we consider an important class of matrices which always have a positive diagonal. An $n \times n$ matrix with nonnegative real entries is called *doubly stochastic* if the row sums and the column sums always equal 1. The next three results are due to König [Koe16].

Lemma 7.4.4. *Every doubly stochastic matrix has a positive diagonal.*

Proof. For doubly stochastic matrices, a positive diagonal is the same as a non-zero diagonal. Thus we may apply Lemma 7.4.2. Now suppose that all non-zero entries of a given set of k columns belong to $r < k$ rows. Denote the matrix determined by these k columns and r rows by B. Then the sum of all entries of B is $= k$ (when added by columns) as well as $\leq r$ (when added by rows), a contradiction. □

We will see that there is a close relationship between doubly stochastic matrices and *permutation matrices*, that is, square matrices which have exactly one entry 1 in each row and column, and all other entries 0.

Theorem 7.4.5 (decomposition theorem). *Let A be an $n \times n$ matrix with nonnegative real entries for which all row sums and all column sums equal some constant s. Then A is a linear combination of permutation matrices with positive real coefficients.*[6]

[6] A strong generalization of Theorem 7.4.5 is proved in [LeLL86].

Proof. Dividing all entries of A by s yields a doubly stochastic matrix A'. By Lemma 7.4.4, A' (and hence A) has a positive diagonal D. Let P be the permutation matrix corresponding to D (that is, P has entry 1 in the cells of D), and let c be the minimum of the entries in D. Then $B = A - cP$ is a matrix with nonnegative real entries and constant row and column sums as well. But B has at least one more entry 0 than A, so that the assertion follows using induction. □

Let us state a simple consequence of our proof of the decomposition theorem, which the reader should compare with Corollary 7.2.6. It – and some generalizations – are important tools in finite geometry, more precisely for the recursive construction of incidence structures; see, for example, the survey[Jun79b].

Corollary 7.4.6 (König's lemma). *Let A be a square matrix with entries 0 and 1 for which all row sums and all column sums equal some constant k. Then A is the sum of k permutation matrices.*

Proof. The assertion follows immediately from the proof of Theorem 7.4.5: in this case, we always have $c = 1$. □

A further immediate consequence of Theorem 7.4.5 is the following classical result due to Birkhoff [Bir46]. We need to recall a fundamental concept from Euclidean geometry: the *convex hull* of n vectors $x_1, \ldots, x_n$ in a real vector space is the set of all linear combinations $x_1c_1 + \ldots + x_nc_n$ with nonnegative coefficients c_i satisfying $c_1 + \ldots + c_n = 1$.

Theorem 7.4.7. *The convex hull of the permutation matrices in $\mathbb{R}^{(n,n)}$ is the set of doubly stochastic matrices.* □

Further theorems from combinatorial matrix theory can be found in the books [Mir71b] and [FoFu62]. Let us mention an interesting strengthening of Lemma 7.4.4 without proof; see [MaMi65].

Result 7.4.8. *Every doubly stochastic $n \times n$ matrix A has a diagonal for which the product of its n entries is at least n^{-n}.* □

The matrix with all entries $1/n$ shows that the bound of Result 7.4.8 is best possible. Summing the products of the entries in D over all diagonals D of a square matrix A gives the *permanent* $\operatorname{per} A$.[7] The *van der Waerden conjecture* [vdW26] suggested a considerably stronger result than 7.4.8; this conjecture remained open for more than fifty years until it was finally proved independently by Egoritsjev and Falikman [Ego81, Fal81]. Proofs can also be found in [Knu81], [Hal86], [Min88], and [vLiWi01].

[7] Note that this function differs from the determinant of A only by the signs of the terms appearing in the sum. Although there exist efficient algorithms for computing determinants, evaluating the permanent of a matrix is NP-hard by a celebrated result of Valiant [Val79a].

Result 7.4.9 (van der Waerden conjecture). *Every doubly stochastic $n \times n$ matrix A satisfies per $A \geq n!/n^n$, with equality only for the matrix with all entries $1/n$.* □

The permanent plays an important role in determining the number of SDR's of a family of sets, and in determining the number of complete matchings of a bipartite graph; see [Mir71b], [Hal86], and [Min78]. As an example, we mention the following interesting application of Result 7.4.9.

Theorem 7.4.10. *Let G be a k-regular bipartite graph with $|S| = |T| = n$. Then G admits at least $n!k^n/n^n$ different perfect matchings.*

Proof. Let A be the 0-1-matrix corresponding to G; that is, $a_{ij} = 1$ if and only if ij is an edge of G. Then the perfect matchings of G correspond to the positive diagonals of A. As $\frac{1}{k}A$ is a doubly stochastic matrix, we have $\mathrm{per}(\frac{1}{k}A) \geq n!/n^n$. Now each diagonal of $\frac{1}{k}A$ has product 0 or $1/k^n$, so that there have to be at least $n!k^n/n^n$ positive diagonals of A. □

Theorem 7.4.10 and its generalizations (see Exercise 7.4.15) are interesting tools in finite geometry; see [Jun79a, Jun79b]. Next we mention two important optimization problems for matrices.

Example 7.4.11 (bottleneck assignment problem). Suppose we are given an $n \times n$ matrix $A = (a_{ij})$ with nonnegative real entries. We want to find a diagonal of A such that the minimum of its entries is maximal. A possible interpretation of this abstract problem is as follows. We need to assign workers to jobs at an assembly line; a_{ij} is a measure of the efficiency of worker i when doing job j. Then the minimum of the entries in a diagonal D is a measure for the efficiency arising from the assignment of workers to jobs according to D.

Problem 7.4.11 can be solved using the above methods as follows. Start with an arbitrary diagonal D whose minimal entry is m, say, and declare all cells (i, j) with $a_{ij} > m$ admissible. Obviously, there will be some diagonal D' with minimal entry $m' > m$ if and only if there is an admissible diagonal for A. This can be checked with complexity $O(|V|^{5/2})$ by determining the cardinality of a maximal matching in the corresponding bipartite graph G. Note that the problem will be solved after at most $O(n^2)$ such steps.[8] The following famous problem – which will be studied extensively in Chapter 14 – can be treated in a similar way.

Example 7.4.12 (assignment problem). Let A be a given square matrix with nonnegative real entries. We require a diagonal of A for which the sum of all its entries is maximal (or minimal). We could interpret this problem

[8] This problem was generalized by Gabow and Tarjan, [GaTa88] who also gave an algorithm with complexity $O((|V| \log |V|)^{1/2}|E|)$. For our classical special case, this yields a complexity of $O((\log |V|)^{1/2}|V|^{5/2})$.

again as finding an assignment of workers to jobs or machines (which are, this time, independent of one another), where the entries of A give the value of the goods produced (or the amount of time needed for a given number of goods to be produced).

As we will see in Chapter 14, the assignment problem can be solved with complexity $O(n^3)$. The *Hungarian algorithm* of Kuhn [Kuh55], which is often used for this task, is based on finding maximal matchings in appropriate bipartite graphs. We close this section with a few exercises, some of which are taken from [MaRe59], [Jun79a], and [FaMi60].

Exercise 7.4.13. Translate Corollary 7.4.6 into the terminology of bipartite graphs.

Exercise 7.4.14. Let A be a doubly stochastic matrix of order n. Then A has a diagonal whose entries have sum at least 1. Hint: Use Result 7.4.8 and the inequality between the arithmetic mean and the geometric mean.

Exercise 7.4.15. Let A be an $m \times n$ 0-1-matrix having row sums tr and column sums $\leq r$. Then A is the sum of r matrices A_i having row sums t and column sums ≤ 1. Hint: Use induction on r by determining an appropriate transversal for the family of sets which contains t copies of each of the sets $T_i = \{j \in \{1, \ldots, n\} \colon a_{ij} = 1\}$ for $i = 1, \ldots, m$.

Exercise 7.4.16. Let A be a 0-1-matrix for which all row and columns sums are at most r. Show that A is the sum of r 0-1-matrices for which all row and columns sums are at most 1. Hint: Translate the claim into the language of bipartite graphs and use Corollary 7.3.12 for a proof by induction.

Exercise 7.4.17. Show that the subspace of $\mathbb{R}^{(n,n)}$ generated by the permutation matrices has dimension $n^2 - 2n + 2$; see Theorem 7.4.7.

7.5 Dissections: Dilworth's theorem

In this section we deal with decomposition theorems for directed graphs and partially ordered sets. Again, we begin with a definition. Let G be a graph or a digraph. A subset X of the vertex set of G is called *independent* or *stable* if no two vertices in X are adjacent; cf. Exercise 2.8.3. The maximal cardinality $\alpha(G)$ of an independent set of G is called the *independence number* of G.[9] Obviously, the complement of an independent set is a vertex cover; this implies the following lemma.

Lemma 7.5.1. *Let G be a graph or a digraph. Then $\alpha(G) + \beta(G) = |V|$.* □

[9] Note that independent sets are the vertex analogue of matchings, which may be viewed as independent sets of edges; hence the notation $\alpha'(G)$ in Section 7.2 for the maximal cardinality of a matching.

For the remainder of this section, let $G = (V, E)$ be a digraph. A *dissection* of G is a set of directed paths in G such that the sets of vertices on these paths form a partition of the vertex set V. One denotes the minimal possible number of paths contained in a dissection by $\Delta(G)$. We have the following major result due to Dilworth [Dil50].

Theorem 7.5.2. *Let G be a transitive acyclic digraph. Then the maximal cardinality of an independent set equals the minimal number of paths in a dissection: $\alpha(G) = \Delta(G)$.*

Proof. Since G is transitive, a directed path can meet an independent set in at most one vertex, and hence $\alpha(G) \leq \Delta(G)$. We shall reduce the reverse inequality to Theorem 7.2.3, an approach introduced in [Ful56]. To this end, we replace each vertex v of G by two vertices v', v'' and construct a bipartite graph H with vertex set $V' \dot\cup V''$, where H contains the edge $v'w''$ if and only if vw is an edge of G.

Let $M = \{v_i'w_i'' : i = 1, \ldots, k\}$ be any matching of cardinality k of H; we claim that M can be used to construct a dissection of G into $n - k$ paths, where $n = |V|$. Note that $v_1w_1, \ldots, v_kw_k$ are edges of G, and that all vertices $v_1, \ldots, v_k$ as well as all vertices $w_1, \ldots, w_k$ are distinct. However, $v_i = w_j$ is possible; in this case, we may join the paths v_iw_i and v_jw_j to form a larger path $v_j \text{—} w_j = v_i \text{—} w_i$. By continuing in this manner (that is, joining paths having the same start or end vertex), we finally obtain c paths whose vertex sets are pairwise disjoint. Suppose these paths have lengths $x_1, \ldots, x_c$. The remaining $n - ((x_1 + 1) + \ldots + (x_c + 1))$ vertices are then partitioned into trivial paths of length 0. Altogether, this yields the desired dissection of G into $n - (x_1 + \ldots + x_c) = n - k$ paths.

In particular, we may choose M as a maximal matching of H, that is, $k = \alpha'(H)$. By Theorem 7.2.3, $\alpha'(H) = \beta(H)$; obviously, $\beta(H) \geq \beta(G)$; and by Lemma 7.5.1, $\alpha(G) = n - \beta(G)$. Hence G can be dissected into

$$n - \alpha'(H) \;=\; n - \beta(H) \;\leq\; n - \beta(G) \;=\; \alpha(G)$$

paths, proving $\Delta(G) \leq \alpha(G)$. □

Dilworth proved Theorem 7.5.2 in the setting of partially ordered sets. Thus let $(M, \preceq)$ be a poset and G the corresponding transitive acyclic digraph; see Example 2.6.1. Then a directed path in G corresponds to a *chain* in $(M, \preceq)$; that is, to a subset of M which is linearly ordered: for any two distinct elements a, b of the subset, $a \prec b$ or $b \prec a$. Similarly, an independent set in G corresponds to an *antichain* of $(M, \preceq)$; that is, to a subset of M consisting of incomparable elements: for any two distinct elements a, b of the subset, neither $a \prec b$ nor $b \prec a$ holds. Then Theorem 7.5.2 translates into the following result stated by Dilworth.

Theorem 7.5.3 (Dilworth's theorem). *Let $(M, \preceq)$ be a partially ordered set. Then the maximal cardinality of an antichain of M is equal to the minimal number of chains into which M can be partitioned.* □

The parameter defined in Theorem 7.5.3 is called the *Dilworth number* of $(M, \preceq)$. Before considering some consequences of Theorem 7.5.3, we return to the proof of Theorem 7.5.2. Obviously, the inequality $\Delta(G) \leq \alpha(G)$ carries over to arbitrary acyclic digraphs.[10] Gallai and Milgram [GaMi60] showed that the graph does not even have to be acyclic for this inequality to hold; however, there is no proof known for this result which uses flow networks or matchings. We leave the proof to the reader as a more demanding exercise:

Exercise 7.5.4 (Gallai-Milgram theorem). Let G be an arbitrary directed graph. Prove the inequality $\Delta(G) \leq \alpha(G)$. Hint: Consider a minimal counterexample.

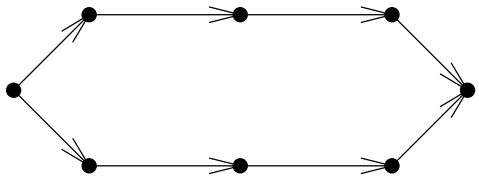


Fig. 7.3. A digraph with $\alpha = 4$ and $\Delta = 2$

Exercise 7.5.5 (Redéi's theorem). A *tournament* is an orientation of a complete graph.[11] Prove that every tournament contains a Hamiltonian path. This result is due to Redéi [Red34].

As promised above, we now derive some consequences of Theorem 7.5.3.

Corollary 7.5.6. *Let $(M, \preceq)$ be a partially ordered set with at least $rs + 1$ elements. Then M contains a chain of cardinality $r + 1$ or an antichain of cardinality $s + 1$.*

Proof. If M does not contain an antichain of cardinality $s+1$, then M can be partitioned into s chains by Theorem 7.5.3. At least one of these chains has to contain at least $r + 1$ elements. □

Corollary 7.5.6 yields a simple proof for the following result originally proved by Erdős and Szekeres [ErSz35].

[10] However, the reverse inequality does not hold for this more general case, as the example in Figure 7.3 shows.

[11] The term *tournament* becomes clear by considering a competition where there are no draws: for example, tennis. Assume that each of n players (or teams, as the case may be) plays against every other one, and that the edge $\{i, j\}$ is oriented as ij if i wins against j. Then an orientation of K_n indeed represents the outcome of a (complete) tournament.

Theorem 7.5.7. *Let $(a_i)_{i=1,\dots,n}$ be a sequence of real numbers, and assume $n \geq r^2 + 1$. Then there exists a monotonic subsequence of length $r + 1$.*

Proof. Put $M = \{(i, a_i) : i = 1, \dots, n\}$ and define a partial ordering $\preceq$ on M as follows:

$$(i, a_i) \preceq (j, a_j) \quad \Longleftrightarrow \quad i \leq j \text{ and } a_i \leq a_j.$$

Now suppose that M contains an antichain with $r + 1$ elements and let $i_1 < i_2 < \ldots < i_{r+1}$ be the first coordinates of these elements. Then the corresponding second coordinates form a strictly decreasing subsequence of length $r + 1$. If there is no such antichain, M has to contain a chain of length $r + 1$, by Corollary 7.5.6; then the second coordinates form an increasing subsequence of length $r + 1$. □

The following well-known result due to Sperner [Spe28] has become the starting-point for a large area of research concerning partially ordered sets: *Sperner Theory*; see the survey [GrKl78] as well as [Gri88] and the monograph [Eng97]. We shall deduce this result by using Dilworth's theorem.

Theorem 7.5.8 (Sperner's lemma). *Let the power set 2^M of a finite set M be partially ordered with respect to inclusion; then the maximal cardinality of an antichain is $N = \binom{n}{\lfloor n/2 \rfloor}$, where $n = |M|$.*

Proof. Obviously, the subsets of cardinality $\lfloor n/2 \rfloor$ form an antichain of cardinality N. To show that there is no antichain having more elements, consider the digraph G with vertex set 2^M corresponding to $(2^M, \subseteq)$. By Theorem 7.5.2, it suffices to partition G into N directed paths. Note that the vertex set of G is partitioned in a natural way into $n + 1$ sets, namely all subsets of M having the same cardinality i for $i = 0, \dots, n$.

Consider the bipartite graph G_k induced by G on the subsets of M of cardinality k or $k + 1$, where $k = 0, \dots, n - 1$. We claim that each of the G_k has a complete matching; we may assume $k + 1 \leq n/2$. Consider an arbitrary collection of j k-subsets of M and note that these subsets are incident with $j(n-k)$ edges in G_k. As each $(k+1)$-subset is on exactly $k+1 \leq n/2$ edges in G_k, these $j(n - k)$ edges have to be incident with at least $j(n - k)/k + 1 \geq j$ distinct $(k+1)$-subsets. By Theorem 7.2.5, G_k indeed has a complete matching. Finally, the edges of the complete matchings of the bipartite graphs G_k $(k = 0, \dots, n - 1)$ can be joined to form the desired directed paths in G. □

A further interesting application of Theorem 7.5.2 treating distributive lattices is given in [Dil50]; we refer to [Aig97] for this so-called *coding theorem* of Dilworth. We pose two more exercises; the first of these is due to Mirsky [Mir71a].

Exercise 7.5.9. Let $(M, \preceq)$ be a partially ordered set. Show that the maximal cardinality of a chain in $(M, \preceq)$ equals the minimal number of antichains into which M can be partitioned. This result is dual to Dilworth's theorem, but its proof is much easier than the proof of Theorem 7.5.2. Hint: Consider the set of maximal elements.

Exercise 7.5.10. Use Dilworth's theorem to derive the marriage theorem.

We remark that our proof of Theorem 7.5.2 is also interesting from an algorithmic point of view, since it allows to calculate the Dilworth number of G by determining a maximal matching in the bipartite graph H, because of $\Delta = n - \alpha'(H)$. Thus Theorem 7.5.2 implies the following result (and its translation to posets which we will leave to the reader).

Corollary 7.5.11. *Let $G = (M, E)$ be a transitive acyclic digraph. Then the maximal cardinality of an independent set of vertices in G – that is, the minimal number of paths in a dissection of G – can be calculated with complexity $O(|M|^{5/2})$.* □

We note that the proof of the theorem of Gallai and Milgram given in the solution for Exercise 7.5.4 is not applicable algorithmically. As this result is stronger than Dilworth's theorem, it would be interesting from the algorithmic point of view to find an alternative proof using the theory of flows, or to reduce the general case to the special case of acyclic digraphs. It is an open problem whether or not such a proof exists.

7.6 Parallelisms: Baranyai's theorem

This section contains an application of the integral flow theorem in finite geometry, namely the theorem of Baranyai [Bar75]. Let X be a given finite set of cardinality n; the elements of X will be called *points*. We denote the set of all t-subsets of X by $\binom{X}{t}$. A *parallelism* of $\binom{X}{t}$ is a partition of $\binom{X}{t}$ whose classes are themselves partitions of X; the classes are called *parallel classes*. Note that a parallelism satisfies the usual Euclidean axiom for parallels: for every point $x \in X$ and for each t-subset Y of X, there is exactly one t-subset Y' which is parallel to Y (that is, contained in the same parallel class as Y) and contains x. Obviously, a parallelism can exist only if t is a divisor of n. It was already conjectured by Sylvester that this condition is sufficient as well. For $t = 3$, the conjecture was proved by Peltesohn [Pel36]; the general case remained open until Baranyai's work. His main idea was to use induction on n; the crucial fact is that this approach requires dealing with an assertion which is much stronger than Sylvester's conjecture (which does not allow an inductive proof), as we shall see later. In fact, Baranyai proved the following result.

Theorem 7.6.1 (Baranyai's theorem). *Let X be a set with n elements, and $A = (a_{ij})_{i=1,\dots,r;j=1,\dots,s}$ a matrix over $\mathbb{Z}_0^+$. Moreover, let $t_1, \dots, t_r$ be integers such that $0 \le t_i \le n$ for $i = 1, \dots, r$. Then there exist subsets A_{ij} of the power set 2^X of X with cardinality a_{ij} satisfying the following two conditions:*

(1) *For each i, $\{A_{i1}, \ldots, A_{is}\}$ is a partition of $\binom{X}{t_i}$.*

(2) *For each j, $A_{1j} \cup \ldots \cup A_{rj}$ is a partition of X.*

if and only if A satisfies the two conditions

(3) $a_{i1} + \ldots + a_{is} = \binom{n}{t_i}$ *for* $i = 1, \ldots, r$,

(4) $t_1 a_{1j} + \ldots t_r a_{rj} = n$ *for* $j = 1, \ldots, s$.

Proof. Trivially, conditions (1) and (2) imply (3) and (4). So suppose conversely that (3) and (4) are satisfied; we have to construct appropriate sets A_{ij}. We do this using induction on n; the induction basis $n = 1$ is trivial. So let $n \neq 1$ and suppose the statement has already been proved for $n - 1$. We sketch the idea of the proof first: suppose we have already found the desired sets A_{ij}. Then, removing some point $x_0 \in X$ from all subsets of X for each i yields partitions of $\binom{X'}{t_i}$ and $\binom{X'}{t_i - 1}$, where $X' := X \setminus \{x_0\}$. Note that x_0 will be removed, for fixed j, from exactly one of the A_{ij}. We want to invert this procedure, which is easier said than done.

Let us define a network $N = (G, c, q, u)$ as follows: G has vertices q (the source); u (the sink); $x_1, \ldots, x_r$; and $y_1, \ldots, y_s$. The edges of G are all the qx_i, with capacity $c(qx_i) = \binom{n-1}{t_i - 1}$; all the $y_j u$, with capacity 1; and all the $x_i y_j$, with capacity 1 or 0 depending on whether $a_{ij} \neq 0$ or $a_{ij} = 0$. Now let f be a flow on N. Then f can have value at most $c(y_1 u) + \ldots + c(y_s u) = s$. We show that a *rational* flow with this value exists. Note

$$c(qx_1) + \ldots + c(qx_r) \;=\; \binom{n-1}{t_1 - 1} + \ldots + \binom{n-1}{t_r - 1} = s;$$

this follows from

$$\sum_{j=1}^{s} (t_1 a_{1j} + \ldots + t_r a_{rj}) \;=\; ns \;=\; \sum_{i=1}^{r} t_i \binom{n}{t_i} = n \sum_{i=1}^{r} \binom{n-1}{t_i - 1},$$

which in turn is a consequence of (3) and (4). Now we define f by

$$f(qx_i) = \binom{n-1}{t_i - 1}, \quad f(y_j u) = 1 \quad \text{and} \quad f(x_i y_j) = \frac{t_i a_{ij}}{n}$$

for $i = 1, \ldots, r$ and $j = 1, \ldots, s$. Condition (4) yields $t_i a_{ij}/n \leq 1 = c(x_i y_j)$, whenever $a_{ij} \neq 0$. Moreover, if f really is a flow, it obviously has value $w(f) = s$. It remains to check the validity of condition (F2) for f:

$$\sum_{e^- = x_i} f(e) \;=\; \sum_{j=1}^{s} f(x_i y_j) \;=\; \sum_{j=1}^{s} \frac{t_i a_{ij}}{n} \;=\; \binom{n}{t_i} \frac{t_i}{n} \;=\; \binom{n-1}{t_i - 1} \;=\; f(qx_i)$$

and

$$\sum_{e^+=y_j} f(e) \;=\; \sum_{i=1}^{r} f(x_i y_j) \;=\; \sum_{i=1}^{r} \frac{t_i a_{ij}}{n} \;=\; 1 \;=\; f(y_j u);$$

these identities follow using (3) and (4), respectively. Summing up, f is indeed a maximal flow on N. By Theorem 6.1.5, there also exists a maximal *integral* flow f' on N; such a flow obviously has to have the form

$$f'(qx_i) = f(qx_i) = \binom{n-1}{t_i - 1},\; f'(y_j u) = f(y_j u) = 1,\; f'(x_i y_j) =: e_{ij} \in \{0, 1\}$$

for $i = 1, \ldots, r$ and $j = 1, \ldots, s$. Moreover, the e_{ij} have to satisfy the following conditions which follow from (F2):

(5) $e_{i1} + \ldots + e_{is} \;=\; \binom{n-1}{t_i - 1}$ for $i = 1, \ldots, r$

and

(6) $e_{1j} + \ldots + e_{rj} \;=\; 1$ for $j = 1, \ldots s$.

Now we put

$$t'_i \;=\; \begin{cases} t_i & \text{for } i = 1, \ldots r \\ t_{i-r} - 1 & \text{for } i = r+1, \ldots 2r \end{cases}$$

and

$$a'_{ij} \;=\; \begin{cases} a_{ij} - e_{ij} & \text{for } i = 1, \ldots, r \\ e_{i-r,j} & \text{for } i = r+1, \ldots 2r \end{cases}$$

for $j = 1, \ldots s$. The condition $0 \le t'_i \le n-1$ holds except if $t_i = n$ or $t_{i-r} = 0$. But these two cases are trivial and may be excluded, as they correspond to partitions of $\{X\}$ and $\{\emptyset\}$. Note $a'_{ij} \ge 0$ for all i, j.

Now we use (5) and (6) to check that the t'_i and the matrix $A' = (a'_{ij})$ satisfy conditions (3) and (4) with $n-1$ instead of n:

$$a'_{i1} + \ldots + a'_{is} = (a_{i1} + \ldots + a_{is}) - (e_{i1} + \ldots + e_{is}) = \binom{n}{t_i} - \binom{n-1}{t_i - 1} = \binom{n-1}{t'_i}$$

for $i = 1, \ldots r$;

$$a'_{i1} + \ldots + a'_{is} \;=\; e_{i-r,1} + \ldots + e_{i-r,s} \;=\; \binom{n-1}{t_{i-r} - 1} \;=\; \binom{n-1}{t'_i}$$

for $i = r+1, \ldots 2r$; and

$$\begin{aligned} a'_{1j} t'_1 + \ldots + a'_{2r,j} t'_{2r} \;&=\; ((a_{1j} - e_{1j}) t_1 + \ldots + (a_{rj} - e_{rj}) t_r) + \\ &\qquad + (e_{1j}(t_1 - 1) + \ldots + e_{rj}(t_r - 1)) \\ &=\; (a_{1j} t_1 + \ldots + a_{rj} t_r) - (e_{1j} + \ldots + e_{rj}) \;=\; n - 1 \end{aligned}$$

for $j = 1, \ldots, s$. Hence the induction hypothesis guarantees the existence of subsets A'_{ij} (for $i = 1, \ldots, 2r$ and $j = 1, \ldots, s$) of $2^{X'}$ satisfying conditions

analogous to (1) and (2). For each j, exactly one of the sets $A'_{r+1,j}, \ldots, A'_{2r,j}$ is nonempty, because of (6). Then this subset contains exactly one $(t_i - 1)$-set, say X_j. We put

$$A_{ij} = \begin{cases} A'_{ij} & \text{for } e_{ij} = 0 \\ A'_{ij} \cup \{X_j \cup \{x_0\}\} & \text{for } e_{ij} = 1 \end{cases}$$

for $i = 1, \ldots, r$ and $j = 1, \ldots, s$. It remains to show that these sets A_{ij} satisfy conditions (1) and (2). Trivially, A_{ij} has cardinality a_{ij}. Moreover, for fixed i, $\{A_{i1}, \ldots, A_{is}\}$ is a partition of $\binom{X}{t_i}$. To see this, let Y be any t_i-subset of X. If Y does not contain x_0, then Y occurs in exactly one of the sets A'_{ij}. If Y contains x_0, then $Y' = Y \setminus \{x_0\}$ occurs in exactly one of the sets $A'_{r+1,j}, \ldots, A'_{2r,j}$, say in $A'_{i+r,j}$; then Y occurs in A_{ij}. Thus (1) holds. Finally, for each j, the set $A'_{1j} \cup \ldots \cup A'_{2r,j}$ is a partition of X'; and as x_0 was added to exactly one of these sets (namely to X_j), condition (2) has to be satisfied as well. □

If we choose $r = 1$, $t_1 = t$, $s = \binom{n-1}{t-1}$, and $a_{1j} = n/t$ (for all j) in Theorem 7.6.1, we obtain the conjecture of Sylvester mentioned at the beginning of this section.

Corollary 7.6.2 (Sylvester's conjecture). *Let X be an n-set and t a positive integer. Then $\binom{X}{t}$ has a parallelism if and only if t divides n.* □

The proof of Theorem 7.6.1 actually yields a method for constructing a parallelism of $\binom{X}{t}$ recursively. However, this approach would not be very efficient because the number of rows of the matrix A doubles with each iteration, so that the complexity is exponential. For $t = 2$, a parallelism is the same as a 1-factorization of the complete graph on X; here Exercise 1.1.2 provides an explicit solution. Beth [Bet74] gave parallelisms for $t = 3$ and appropriate values of n (using finite fields); see also [BeJL99], §VIII.8. No such series of parallelisms are known for larger values of t. The interesting monograph [Cam76] about parallelisms of complete designs (those are exactly the parallelisms defined here) should be mentioned in this context. Also, we remark that in finite geometry, parallelisms are studied for several other kinds of incidence structures, for example in Kirkman's school girl problem (see Section 7.2); we refer the reader to [BeJL99].

7.7 Supply and demand: The Gale-Ryser theorem

In the final section of this chapter, we consider a further application of network flow theory in optimization, namely the *supply and demand problem*.[12] Let (G, c) be a network, and let X and Y be disjoint subsets of the vertex set V.

[12] A somewhat more general problem will be the subject of Chapter 11.

The elements x of X are considered to be *sources*, and the vertices y in Y are interpreted as *sinks*. With each source x, we associate a *supply* $a(x)$, and with each sink y a *demand* $d(y)$ – intuitively, we may think of companies producing a certain product and customers who want to buy it. A *feasible flow* on (G, c) is a mapping $f\colon E \to \mathbb{R}_0^+$ satisfying the following conditions:

(ZF 1) $0 \leq f(e) \leq c(e)$ for all $e \in E$;

(ZF 2) $\sum_{e^-=x} f(e) - \sum_{e^+=x} f(e) \leq a(x)$ for all $x \in X$;

(ZF 3) $\sum_{e^+=y} f(e) - \sum_{e^-=y} f(e) = d(y)$ for all $y \in Y$;

(ZF 4) $\sum_{e^+=v} f(e) = \sum_{e^-=v} f(e)$ for all $v \in V \setminus (X \cup Y)$.

Thus the amount of flow coming out of a source cannot be larger than the corresponding supply $a(x)$, and the amount of flow going into a sink has to equal the corresponding demand $d(y)$. For all other vertices (which are often called *intermediate nodes* or *transshipment nodes*), the amount of flow going into that vertex has to be the same as the amount of flow coming out of it; this agrees with the flow conservation condition (F2). We have the following result due to Gale [Gal57].

Theorem 7.7.1 (supply and demand theorem). *For a given supply and demand problem (G, c, X, Y, a, d), there exists a feasible flow if and only if the following condition is satisfied:*

$$c(S,T) \;\geq\; \sum_{y \in Y \cap T} d(y) - \sum_{x \in X \cap T} a(x) \quad \textit{for each cut } (S,T) \textit{ of } G.^{13} \tag{7.4}$$

Proof. We reduce the existence problem for feasible flows to usual network flows. To this end, we add two new vertices to G, namely the source s and the sink t; and we also add all edges sx (for $x \in X$) with capacity $c(sx) = a(x)$, and all edges yt (for $y \in Y$) with capacity $c(yt) = d(y)$. This yields a standard flow network N. It is easy to see that the original problem admits a feasible flow if and only if there exists a flow on N which saturates all edges yt; that is, if and only if the maximal value of a flow on N equals the sum w of the demands $d(y)$. Using Theorem 6.1.6, this means that there exists a feasible flow if and only if each cut in N has capacity at least w. Note that a cut in N has the form $(S \cup \{s\}, T \cup \{t\})$, where (S, T) is a cut in G. Hence we get the condition

$$c(S \cup \{s\}, T \cup \{t\}) \;=\; c(S,T) + \sum_{x \in X \cap T} a(x) + \sum_{y \in Y \cap S} d(y) \;\geq\; \sum_{y \in Y} d(y),$$

and the assertion follows. □

[13] In contrast to our former definition, $S = \emptyset$ or $T = \emptyset$ are allowed here.

Let us apply Theorem 7.7.1 to derive necessary and sufficient conditions for the existence of a bipartite graph G with vertex set $V = S \dot\cup T$ and given degree sequences $(p_1, \ldots, p_m)$ for the vertices in S, and $(q_1, \ldots, q_n)$ for the vertices in T. We may assume $q_1 \geq q_2 \geq \ldots \geq q_n$; we will see that this assumption is quite helpful. An obvious necessary condition for the existence of such a graph is $p_1 + \ldots + p_m = q_1 + \ldots + q_n$; however, this condition is not sufficient.

Exercise 7.7.2. Show that there is no bipartite graph with degree sequences $(5, 4, 4, 2, 1)$ and $(5, 4, 4, 2, 1)$.

The following theorem of Gale [Gal57] and Ryser [Rys57] gives the desired criterion:

Theorem 7.7.3 (Gale-Ryser theorem). *Let $(p_1, \ldots, p_m)$ and $(q_1, \ldots, q_n)$ be two sequences of nonnegative integers satisfying the conditions*

$$q_1 \geq q_2 \geq \ldots \geq q_n \quad \text{and} \quad p_1 + \ldots + p_m = q_1 + \ldots + q_n.$$

Then there exists a bipartite graph G with vertex set $V = X \dot\cup Y$ and degree sequences $(p_1, \ldots, p_m)$ on X and $(q_1, \ldots, q_n)$ on Y if and only if the following condition holds:

$$\sum_{i=1}^{m} \min(p_i, k) \;\geq\; \sum_{j=1}^{k} q_j \quad \text{for } k = 1, \ldots, n. \tag{7.5}$$

Proof. Let $X = \{x_1, \ldots, x_m\}$ and $Y = \{y_1, \ldots, y_n\}$. We define a supply and demand problem as follows. The network (G, c) contains all edges $x_i y_j$ with capacity $c(x_i y_j) = 1$. Moreover, with x_i we associate the supply $a(x_i) = p_i$, and with y_j we associate the demand $d(y_j) = q_j$. Obviously, the existence of a feasible flow for (G, c) is equivalent to the existence of a bipartite graph with vertex set $V = X \dot\cup Y$ having the prescribed degree sequences: the edges with non-zero flow in the network are precisely the edges of G. Thus we need to check that condition (7.4) in Theorem 7.7.1 is equivalent to (7.5).

For each subset U of V, put $U' = \{i : x_i \in U\}$ and $U'' = \{j : y_j \in U\}$. Then $c(S, T) = |S'||T''|$, where $T := V \setminus S$. First, suppose there exists a feasible flow. Then (7.4) implies

$$|S'||T''| \;\geq\; \sum_{j \in T''} q_j - \sum_{i \in T'} p_i \quad \text{for all } S \subset V. \tag{7.6}$$

Choosing $S = \{x_i : p_i > k\} \cup \{y_{k+1}, \ldots, y_n\}$, (7.6) becomes

$$k \times \big|\{i : p_i > k\}\big| \;\geq\; \sum_{j=1}^{k} q_j - \sum_{p_i \leq k} p_i.$$

This implies (7.5) noting the following fact: for $p_i \leq k$, we have $p_i = \min(p_i, k)$; and for $p_i > k$, we have $k = \min(p_i, k)$.

Conversely, suppose that condition (7.5) is satisfied, and let S be an arbitrary subset of V. Consider the cut (S, T), where $T = V \setminus S$. With $k = |T''|$, we get

$$\begin{aligned} c(S,T) &= \sum_{i \in S'} k \geq \sum_{i \in S'} \min(p_i, k) \geq \sum_{j=1}^{k} q_j - \sum_{i \in T'} \min(p_i, k) \\ &\geq \sum_{j \in T''} q_j - \sum_{i \in T'} p_i = \sum_{y \in Y \cap T} d(y) - \sum_{x \in X \cap T} a(x). \end{aligned}$$

Thus (7.5) indeed implies (7.4). □

Actually, Ryser stated and proved Theorem 7.7.3 in the language of 0-1-matrices. With any bipartite graph $G = (X \dot{\cup} Y, E)$, we associate – as usual – a matrix $M = (m_{xy})_{x \in X, y \in Y}$, where $m_{xy} = 1$ if $xy \in E$ and $m_{xy} = 0$ otherwise. Conversely, each 0-1-matrix yields a bipartite graph. Then the degree sequence on X corresponds to the sequence of row sums of M, and the degree sequence on Y corresponds to the sequence of column sums of M. In this way, Theorem 7.7.3 translates into the following criterion for the existence of a 0-1-matrix with given row and column sums.

Theorem 7.7.4. *Let $(p_1, \ldots, p_m)$ and $(q_1, \ldots, q_n)$ be two sequences of non-negative integers satisfying the conditions $q_1 \geq q_2 \geq \ldots \geq q_n$ and $q_1 + \ldots + q_n = p_1 + \ldots + p_m$. Then there exists an $m \times n$ 0-1-matrix with row sums $(p_1, \ldots, p_m)$ and column sums $(q_1, \ldots, q_n)$ if and only if condition (7.4) in Theorem 7.7.3 holds.* □

A different proof of Theorems 7.7.3 and 7.7.4 using the methods of transversal theory can be found in [Mir71b].

Exercise 7.7.5. Suppose we are given a supply and demand problem where the functions c, a, and d are integral. If there exists a feasible flow, is there an integral feasible flow as well?

8

Connectivity and Depth First Search

How beautiful the world would be if there were a rule for getting around in labyrinths.
Umberto Eco

We have already encountered the notions of connectivity, strong connectivity, and k-connectivity; and we know an efficient method for determining the connected components of a graph: breadth first search. In the present chapter, we mainly treat algorithmic questions concerning k-connectivity and strong connectivity. To this end, we introduce a further important strategy for searching graphs and digraphs (besides BFS), namely *depth first search*. In addition, we present various theoretical results, such as characterizations of 2-connected graphs and of edge connectivity.

8.1 k-connected graphs

In Section 7.1, we defined the connectivity $\kappa(G)$ of a graph and introduced k-connected graphs. As Exercise 7.1.7 shows, these notions are intimately related to the existence of vertex disjoint paths. This suggests a further definition: for any two vertices s and t of a graph G, we denote by $\kappa(s,t)$ the maximal number of vertex disjoint paths from s to t in G. By Menger's theorem, $\kappa(s,t)$ equals the minimal cardinality of a vertex separator for s and t whenever s and t are non-adjacent. Using this notation, we may re-state the result in Exercise 7.1.7 as follows.

Theorem 8.1.1 (Whitney's theorem). *A graph G is k-connected if and only if $\kappa(s,t) \geq k$ for any two vertices s and t of G. Hence*

$$\kappa(G) = \min \{\kappa(s,t) \colon s,t \in V\}. \tag{8.1}$$

Exercise 8.1.2. Show that a k-connected graph on n vertices contains at least $\lceil kn/2 \rceil$ edges. (Note that this bound is tight; see [Har62].)

Exercise 8.1.3. Let $G = (V,E)$ be a k-connected graph, T a k-subset of V, and $s \in V \setminus T$. Show that there exists a set of k paths with start vertex s and end vertex in T for which no two of these paths share a vertex other than s.

We will soon present an algorithm which determines the connectivity of a given graph. First we apply Exercise 8.1.3 to the existence problem for Hamiltonian cycles; the following sufficient condition is due to Chvátal and Erdös [ChEr72].

Theorem 8.1.4. *Let G be a k-connected graph, where $k \geq 2$. If G contains no independent set of cardinality $k+1$ – that is, if $\alpha(G) \leq k$ – then G is Hamiltonian.*

Proof. As G is 2-connected, G has to contain cycles.[1] Let C be a cycle of maximal length m. Assume $m \leq k$. Then G is also m-connected, and we can apply Exercise 8.1.3 to the vertex set of C (as the set T) and an arbitrary vertex $s \notin T$. Replacing one edge $e = uv$ of C with the resulting vertex disjoint paths from s to u and v, respectively, we obtain a cycle of length $> m$. This contradiction shows $m > k$.

Now suppose that C is not a Hamiltonian cycle. Then there exists a vertex $s \notin C$. Again by Exercise 8.1.3, there exist k paths W_i $(i = 1, \ldots, k)$ with start vertex s and end vertex t_i on C which are pairwise disjoint except for s. Moreover, we may assume that t_i is the only vertex W_i has in common with C. Now consider C as a directed cycle – the choice of the orientation does not matter – and denote the successor of t_i on C by u_i. If s is adjacent to one of the vertices u_i, we may replace the edge $t_i u_i$ of C by the path from t_i to s followed by the edge su_i. Again, this yields a cycle of length $> m$, a contradiction. Hence s cannot be adjacent to any of the vertices u_i. As $\alpha(G) \leq k$, the $(k+1)$-set $\{s, u_1, \ldots, u_k\}$ cannot be independent, and hence G contains an edge of the form $u_i u_j$. Then, by replacing the edges $t_i u_i$ and $t_j u_j$ of C by the edge $u_i u_j$ and the paths from s to t_i and from s to t_j, we get a cycle of length $> m$ once again. This final contradiction shows that C has to be a Hamiltonian cycle. □

Corollary 8.1.5. *Assume that the closure $[G]$ of a graph G satisfies the condition $\alpha([G]) \leq \kappa([G])$. Then G is Hamiltonian.*

Proof. This follows immediately from Theorems 1.4.1 and 8.1.4. □

Exercise 8.1.6. Show that Theorem 8.1.4 is best possible by constructing (for each choice of $\kappa(G)$) a graph G with $\alpha(G) = \kappa(G) + 1$ which is not Hamiltonian.

Now we turn to the problem of efficiently determining the connectivity of a given graph. By Theorem 8.1.1, it suffices to determine the maximal number of vertex disjoint paths between any two vertices of G. Corollary 7.1.5 states that this can be done with complexity $O(|V|^{1/2}|E|)$ for an arbitrary pair of vertices, so that we have a total complexity of $O(|V|^{5/2}|E|)$. If G is not

[1] For the time being, we leave it to the reader to prove this claim; alternatively, see Theorem 8.3.1.

a complete graph, we actually need to examine only non-adjacent pairs of vertices, as we will see in the proof of Theorem 8.1.9. We shall present a simple algorithm due to Even and Tarjan [EvTa75] which achieves a slightly better complexity.

Exercise 8.1.7. Design an algorithm for determining the maximal value of a flow on a 0-1-network (MAX01FLOW); and an algorithm for calculating the maximal number of vertex disjoint paths between two given vertices s and t in a graph or digraph (PATHNR). Hint: Use the results of Sections 6.5 and 7.1.

Algorithm 8.1.8. Let $G = (V, E)$ be a graph on n vertices. The algorithm calculates the connectivity of G.

Procedure KAPPA(G;kappa)

(1) $n \leftarrow |V|$, $k \leftarrow 0$, $y \leftarrow n-1$, $S \leftarrow V$;
(2) **repeat**
(3) choose $v \in S$ and remove v from S;
(4) **for** $w \in S \setminus A_v$ **do** PATHNR$(G, v, w; x)$; $y \leftarrow \min\{y, x\}$ **od**
(5) $k \leftarrow k+1$
(6) **until** $k > y$;
(7) kappa $\leftarrow y$

Theorem 8.1.9. *Let $G = (V, E)$ be a connected graph. Then Algorithm 8.1.8 calculates with complexity $O(|V|^{1/2}|E|^2)$ the connectivity of G.*

Proof. If G is a complete graph K_n, the algorithm terminates (after having removed all n vertices) with kappa $= y = n-1$. Now assume that G is not complete. During the **repeat**-loop, vertices $v_1, v_2, \ldots, v_k$ are chosen one by one until the minimum γ of all values $\kappa(v_i, w_i)$ is less than k, where w_i runs through the vertices which are not adjacent to v_i; then $k \geq \gamma + 1 \geq \kappa(G) + 1$. By definition, there exists a vertex separator T for G of cardinality $\kappa(G)$. As $k \geq \kappa(G) + 1$, there is at least one vertex $v_i \notin T$. Now $G \setminus T$ is not connected; hence there exists a vertex v in $G \setminus T$ so that each path from v to v_i meets the set T. In particular, v_i and v cannot be adjacent; thus $\gamma \leq \kappa(v_i, v) \leq |T| = \kappa(G)$, so that $\gamma = \kappa(G)$. This shows that the algorithm is correct. The complexity is $O(\kappa(G)|V|^{3/2}|E|)$: during each of the $\kappa(G)$ iterations of the **repeat**-loop, the procedure PATHNR is called $O(|V|)$ times, and each of these calls has complexity $O(|V|^{1/2}|E|)$. Trivially, $\kappa(G) \leq \deg v$ for each vertex v. Using the equality $\sum_v \deg v = 2|E|$, we get

$$\kappa(G) \;\leq\; \min\{\deg v\colon\, v \in V\} \;\leq\; 2|E|/|V|,$$

which yields the desired complexity $O(|V|^{1/2}|E|^2)$. □

As we have seen, it takes a considerable amount of work to determine the exact value of $\kappa(G)$. In practice, one is often satisfied with checking whether G is at least k-connected. For $k = 1$, this can be done with complexity $O(|E|)$

using BFS; see Section 3.3. For $k = 2$, we shall present an algorithm in Section 8.3 which also has complexity $O(|E|)$. Even for $k = 3$, it is possible to achieve a complexity of $O(|E|)$, albeit with considerably more effort; see [HoTa73]. There is an algorithm having complexity $O(k|V||E|)$ provided that $k \leq |V|^{1/2}$, see [Eve77, Eve79]; in particular, it is possible to check with complexity $O(|V||E|)$ whether a graph is k-connected when k is fixed. This problem is also treated in [Gal80] and in [LiLW88], where an unusual approach is used. For the purpose of designing communication networks it is of interest to find a k-connected subgraph of minimal weight in a directed complete graph; for this problem, we refer to [BiBM90] and the references given there.

8.2 Depth first search

In this section we treat an important method for searching graphs and digraphs, which will be used repeatedly throughout the present chapter. Recall that the BFS in Algorithm 3.3.1 examines a graph G in a *breadth–first* fashion: vertices which have larger distance to the start vertex s are examined later than those with smaller distance to s. In contrast, *depth first search* follows paths as far as possible: from a vertex v already reached, we proceed to any vertex w adjacent to v which was not yet visited; then we go on directly from w to another vertex not yet reached etc., as long as this is possible. (If we cannot go on, we backtrack just as much as necessary.) In this way, one constructs maximal paths starting at some initial vertex s. This idea seems to go back to M. Tremaux who suggested it in 1882 as a method to traverse mazes; see [Luc82] and [Tar95]. The following version is taken from the fundamental paper by Tarjan [Tar72].

Algorithm 8.2.1 (depth first search, DFS). Let $G = (V, E)$ be a graph and s a vertex of G.

Procedure DFS$(G, s; nr, p)$

(1) **for** $v \in V$ **do** $nr(v) \leftarrow 0$; $p(v) \leftarrow 0$ **od**
(2) **for** $e \in E$ **do** $u(e) \leftarrow$ false **od**
(3) $i \leftarrow 1$; $v \leftarrow s$; $nr(s) \leftarrow 1$;
(4) **repeat**
(5) **while** there exists $w \in A_v$ with $u(vw) =$ false **do**
(6) choose some $w \in A_v$ with $u(vw) =$ false; $u(vw) \leftarrow$ true;
(7) **if** $nr(w) = 0$ **then** $p(w) \leftarrow v$; $i \leftarrow i + 1$; $nr(w) \leftarrow i$; $v \leftarrow w$ **fi**
(8) **od**
(9) $v \leftarrow p(v)$
(10) **until** $v = s$ **and** $u(sw) =$ true for all $w \in A_s$

The algorithm labels the vertices with numbers nr according to the order in which they are reached; $p(w)$ is the vertex from which w was accessed.

Theorem 8.2.2. *Each edge in the connected component of s is used exactly once in each direction during the execution of Algorithm 8.2.1. Hence Algorithm 8.2.1 has complexity $O(|E|)$ for connected graphs.*

Proof. We may assume that G is connected. First, we give a more precise meaning to the assertion by showing that the DFS constructs a walk in G beginning in s. In step (6), an edge $e = vw$ (where initially $v = s$) is used to move from v to w; if $nr(w) = 0$, v is replaced by w. If $nr(w) \neq 0$, e is used immediately in the opposite direction to backtrack from w to v, and the algorithm proceeds (if possible) with another edge incident with v which was not yet used. If there is no such edge available – that is, if all edges incident with v have been used at least once – the edge $p(v)v$ which was used to reach v from $p(v)$ is traversed in the opposite direction to backtrack again, and v is replaced by $p(v)$. Thus the algorithm indeed constructs a walk in G.

Now we show that no edge can be used twice in the same direction so that the walk is in fact a trail in $\vec{G}$. Suppose this claim is false; then there is an edge $e = vw$ which is used twice in the same direction. We may assume that e is the first such edge and that e is used from v to w. As each edge is labelled *true* in step (6) when it is used first, $u(e)$ must have value *true* when e is used the second time; hence this event occurs during an execution of step (9). But then $w = p(v)$, and all edges incident with v have to be labelled *true* already, because of the condition in (5). Thus the walk must have left v at least $\deg v$ times before e is used for the second time. This means that the walk must have arrived at v at least $\deg v + 1$ times; therefore some edge of the form uv must have been used twice from u to v before, a contradiction.

The preceding considerations imply that the algorithm terminates. It remains to show that each edge of G is used in both possible directions. Let S be the set of all vertices v for which each edge incident with v is used in both directions. When the algorithm terminates, it must have reached a vertex v with $p(v) = 0$ for which there is no edge incident with v which is labelled with *false* (because of (10)). This can only happen for $v = s$; moreover, all edges incident with s must have been used to move from s to their other end vertex. But then all these $\deg s$ edges must also have been used to reach s, since none of them was used twice in the same direction. This means $s \in S$.

We now claim $S = V$. Suppose otherwise. As G is connected, there exist edges connecting S to $V \setminus S$. Let $e = vw$ be the edge with $v \in S$ and $w \in V \setminus S$ which is used first during the algorithm. Note that every edge connecting some vertex of S and some vertex of $V \setminus S$ is used in both directions, by the definition of S. As we reach vertex w for the first time when we use e from v to w, $nr(w) = 0$ at that point of time. Then, in step (7), we set $v = p(w)$, and e is labelled *true*. Now we can only use e again according to step (9), that is, from w to v. At that point, all edges incident with w must have been labelled *true*. As each edge incident with w can only be used at most once in each direction, each of these edges must have been used in both directions, so that $w \in S$, a contradiction. □

Theorem 8.2.2 shows that depth first search is indeed a possible strategy for finding the exit of a maze, provided that it is possible to label edges – that is, paths in the maze – which have been used already; see Exercise 8.2.6. In the next section, we shall use a refined version of DFS for studying 2-connected graphs. Now we give a much simpler application.

Theorem 8.2.3. *Let G be a connected graph and s a vertex of G. Determine the function p by a call of* $\mathrm{DFS}(G, s; nr, p)$. *Then the digraph on V with edges $p(v)v$ is a spanning arborescence for G with root s.*

Proof. Denote the digraph in question by T. As each vertex v of G is reached for the first time during the DFS via the edge $p(v)v$, $|T|$ is obviously connected. More precisely, the sequence $v = v_0, v_1, v_2, \ldots$ with $v_{i+1} = p(v_i)$ for $v_i \neq s$ yields a path from s to v in T. Thus s is a root of T. Moreover, T contains exactly $|V| - 1$ edges (note that $p(s) = 0$ is not a vertex of G). Hence $|T|$ is a tree by Theorem 1.2.8. □

Hence we may use either BFS (as in Chapter 3.3) or DFS to check with complexity $O(|E|)$ whether a given graph G is connected and – if this is the case – to construct a spanning tree rooted at s.

The edges $p(v)v$ contained in the spanning arborescence of Theorem 8.2.3 are called *tree edges*, whereas all other edges of G are called *back edges*; the next result will explain this terminology. Let us call a vertex u in a directed tree an *ancestor* of some other vertex v if there exists a directed path from u to v in T; similarly, u is a *descendant* of v if there is a directed path from v to u.

Lemma 8.2.4. *Let G be a connected graph, and let T be a spanning arborescence of G determined by a call of* $\mathrm{DFS}(G, s; nr, p)$. *Moreover, let $e = vu$ be a back edge of G. Then u is an ancestor or a descendant of v in T.*

Proof. We may assume $nr(v) < nr(u)$; that is, during the DFS u is reached after v. Note that all edges incident with v have to be traversed starting from v and then labelled *true* (in step (5)) before the algorithm can backtrack from v according to step (9); in particular, this holds for e. As u is not a direct descendant of v (because otherwise $v = p(u)$ so that e would be a tree edge), u must have been labelled before e was examined. This means that u is an indirect descendant of v. □

Example 8.2.5. We consider the graph G of Figure 8.1 and perform a DFS beginning at s. To make the algorithm deterministic, we choose the edges in step (6) according to alphabetical order of their end vertices. Then the vertices are reached in the following order: s, a, b, c, d, e, f, g. After that, the algorithm backtracks from g to f, to e, and then to d. Now h is reached and the algorithm backtracks to d, c, b, a, and finally to s. The directed tree T constructed by the DFS is shown in Figure 8.2.

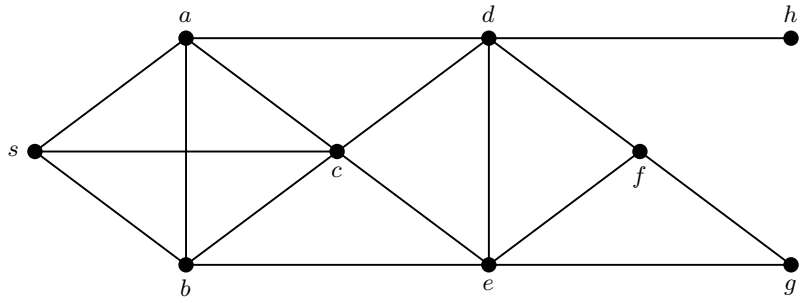


Fig. 8.1. A graph G

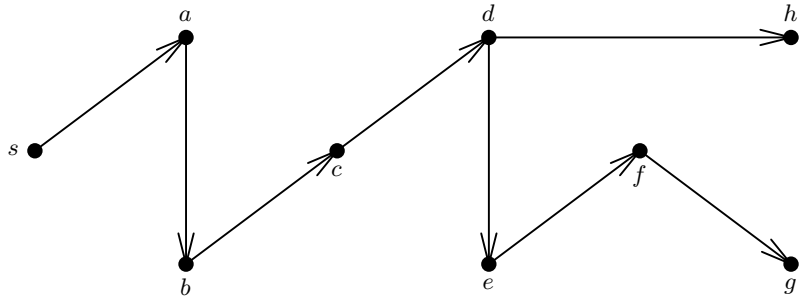


Fig. 8.2. DFS tree for G

In comparison, the BFS algorithm of Section 3.3 treats the vertices in the order $s, a, b, c, d, e, f, h, g$; the corresponding tree T' was already given in Figure 3.5. Note that the distance $d(s, x)$ in T' is equal to the corresponding distance in G, whereas this is not true in T: here vertex g has distance 7 from s (the maximal distance from s occurring in T). This illustrates the phenomenon that the DFS indeed tries to move as deeply into the graph as possible.

Exercise 8.2.6. Describe a way of associating a graph with a given maze which allows us to find a path through the maze via a depth first search. Apply this approach to the maze depicted in Figure 8.3. This task is somewhat lengthy, but really instructive.

8.3 2-connected graphs

A *cut point* or *articulation point* of a graph G is a vertex v such that $G \setminus v$ has more connected components than G. According to Definition 7.1.6, a connected graph with at least three vertices either contains a cut point or is

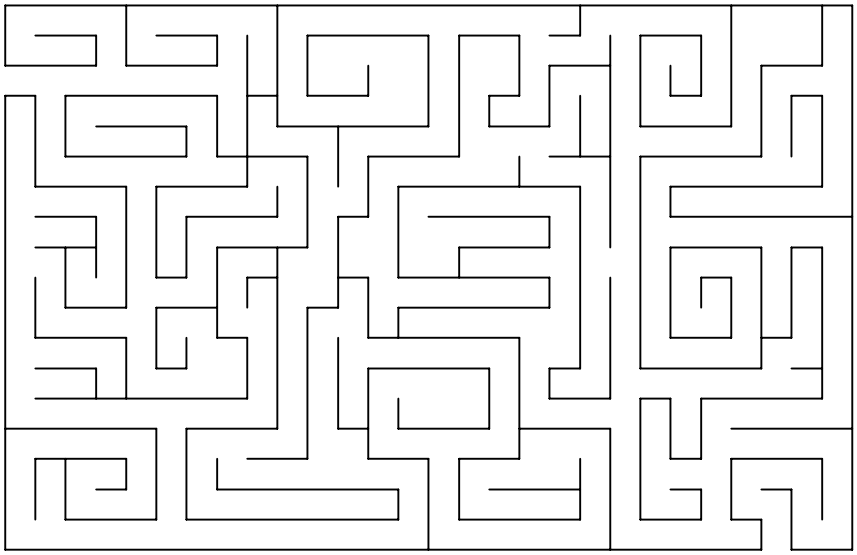


Fig. 8.3. A maze

2-connected. A connected graph containing cut points is said to be *separable*. The maximal induced subgraphs of a graph G which are not separable are called the *blocks* or *biconnected components* of G.

Recall that the connected components of a graph form a partition of its vertex set. The analogous statement for blocks does not hold in general. For example, the graph given in Figure 8.1 has blocks $\{s, a, b, c, d, e, f, g\}$ and $\{d, h\}$. If c is a cut point of a connected graph G, then $V \setminus c$ can be partitioned into sets $V_1 \dot{\cup} \ldots \dot{\cup} V_k$ such that two vertices a and b are in the same part of the partition if and only if they are connected by a path not containing c. Thus no block can contain vertices from more than one of the V_i; in particular, two blocks intersect in at most one vertex, and this vertex has to be a cut point. Let us mention another useful observation: every cycle has to be contained in some block.

We now give some conditions equivalent to 2-connectedness; these are due to Whitney [Whi32b].

Theorem 8.3.1. *Let G be a graph with at least three vertices and with no isolated vertices. Then the following conditions are equivalent:*

(1) *G is 2-connected.*
(2) *For every pair vertices of G, there exists a cycle containing both of them.*
(3) *For each vertex v and for each edge e of G, there exists a cycle containing both v and e.*

(4) *For every pair of edges of G, there exists a cycle containing both of them.*
(5) *For every pair of vertices $\{x, y\}$ and for each edge e of G, there exists a path from x to y containing e.*
(6) *For every triple of vertices (x, y, z) of G, there exists a path from x to y containing z.*
(7) *For every triple of vertices (x, y, z) of G, there exists a path from x to y not containing z.*

Proof.

(1) $\Leftrightarrow$ (2) : If G is 2-connected, Theorem 8.1.1 implies that any two vertices are connected by two vertex disjoint paths; the union of these paths yields the desired cycle containing both vertices. Conversely, a graph satisfying condition (2) obviously cannot contain any cut points.

(1) $\Rightarrow$ (3) : Let $e = uw$; we may assume $v \neq u, w$. We subdivide e by a new vertex x; that is, we replace e by the edges ux and xw to get a new graph G'; compare Section 1.5. As G satisfies (2) as well, G' cannot contain any cut points, that is, G' is likewise 2-connected. Hence G' satisfies (2), and there is a cycle containing v and x in G'; then the corresponding cycle in G has to contain v and e.

(3) $\Rightarrow$ (2) : Let u and v be two vertices of G. As G does not contain any isolated vertices, there exists an edge e incident with u. If (3) holds, there exists a cycle containing e and v; this cycle also contains u and v.

(1) $\Rightarrow$ (4) : Similar to (1) $\Rightarrow$ (3).

(4) $\Rightarrow$ (2) : Similar to (3) $\Rightarrow$ (2).

(1) $\Rightarrow$ (5) : Let G' be the graph we get by adding an edge $e' = xy$ to G (if x and y are not adjacent in G in the first place). Obviously G' is 2-connected as well, so that (4) implies the existence of a cycle in G' containing e and e'. Removing e' from this cycle yields the desired path in G.

(5) $\Rightarrow$ (6) : Choose an edge e incident with z. By (5), there is a path from x to y containing e – and then z as well, of course.

(6) $\Rightarrow$ (7) : As (6) holds for any three vertices of G, there exists a path from x to z containing y. The first part of this path (the part from x to y) is the desired path.

(7) $\Rightarrow$ (1) : If (7) holds, G can obviously not contain any cut points. □

Exercise 8.3.2. Let G be a connected graph with at least two vertices. Show that G contains at least two vertices which are not cut points. Is this bound tight?

Exercise 8.3.3. This exercise deals with some results due to Gallai [Gal64]. Given any graph G, we define the *block-cutpoint graph* $bc(G)$ as follows. The vertices of $bc(G)$ are the blocks and the cut points of G; a block B and a cut point c of G are adjacent in $bc(G)$ if and only if c is contained in B. Show that the following assertions hold.

(a) If G is connected, $bc(G)$ is a tree.

(b) For each vertex v of G, let $b(v)$ denote the number of blocks containing v. Moreover, let $b(G)$ be the number of blocks of G, and denote the number of connected components of G by p. Then

$$b(G) \;=\; p + \sum_{v} (b(v) - 1).$$

(c) For each block B, let $c(B)$ be the number of cut points contained in B, and let $c(G)$ be the number of all cut points of G. Then

$$c(G) \;=\; p + \sum_{B} (c(B) - 1).$$

(d) $b(G) \;\geq\; c(G) + 1$.

Exercise 8.3.4. Let G be a connected graph with r cut points. Show that G has at most $\binom{n-r}{2} + r$ edges, and construct a graph where this bound is tight [Ram68]. Hint: Use the number k of blocks of G and part (d) of Exercise 8.3.3; also, derive a formula for the sum of the cardinalities of the blocks from part (b) of 8.3.3.

For the remainder of this section, let $G = (V, E)$ be a connected graph. Suppose we have constructed a spanning arborescence T for G with root s by a call of $\mathrm{DFS}(G, s; nr, p)$; see Theorem 8.2.3. We will use the functions nr and p for determining the cut points of G (and hence the blocks). We also require a further function L (for *low point*) defined on V: for a given vertex v, consider all vertices u which are accessible from v by a path (possibly empty) which consists of a directed path in T followed by at most one back edge; then $L(v)$ is the minimum of the values $nr(u)$ for all these vertices u.

Example 8.3.5. Let G be the graph of Example 8.2.5; see Figure 8.1. In Figure 8.4, the vertices of G are labelled with the numbers they are assigned during the DFS; the numbers in parentheses are the values of the function L. The thick edges are the edges of the directed tree constructed by the DFS.

Note that the easiest way of computing the values $L(i)$ is to begin with the leaves of the tree; that is, to treat the vertices ordered according to decreasing DFS numbers. In Algorithm 8.3.8, we will see how the function L may be calculated *during* the DFS. The following result of Tarjan [Tar72] shows why this function is important.

Lemma 8.3.6. *Let G be a connected graph, s a vertex of G, and T the spanning arborescence of G determined by a call of* $\mathrm{DFS}(G, s; nr, p)$. *Moreover, let u be a vertex of G distinct from s. Then u is a cut point if and only if there is a tree edge $e = uv$ satisfying $L(v) \geq nr(u)$, where L is the function defined above.*

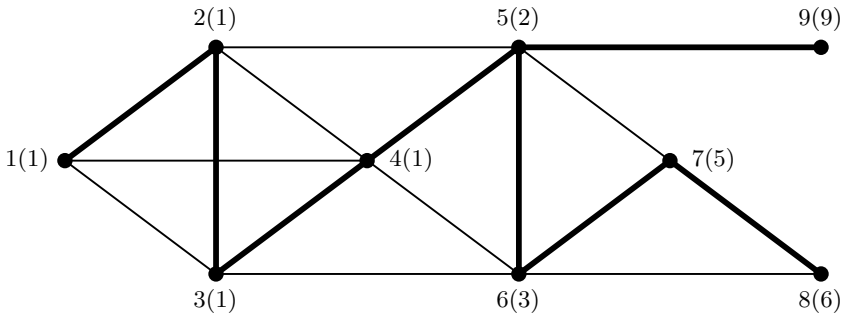


Fig. 8.4. Labels assigned during the DFS and function L

Proof. First suppose that u is a cut point of G. Then there is a partition $V \setminus u = V_1 \dot\cup \ldots \dot\cup V_k$ (where $k \geq 2$) for which all paths connecting two vertices in distinct components of the partition have to pass through u. We may assume $s \in V_1$. Let e be the first tree edge of the form $e = uv$ traversed during the DFS for which $v \notin V_1$ holds, say $v \in V_2$. As there are no edges connecting a vertex in V_2 with a vertex in $V \setminus (V_2 \cup \{u\})$ and as all vertices which are accessible from v by tree edges are again in V_2 (and are therefore reached at a later point of the algorithm than u), we conclude $L(v) \geq nr(u)$.

Conversely, let $e = uv$ be a tree edge with $L(v) \geq nr(u)$. Denote the set of all vertices on the path from s to u in T by S (including s, but not u), and let T' be the part of T having root v; that is, T' consists of the descendants of v. By Lemma 8.2.4, there cannot be an edge connecting a vertex of T' with a vertex of $V \setminus (S \cup T' \cup \{u\})$. Moreover, there are no edges of the form xy with $x \in T'$ and $y \in S$: such an edge would be a back edge, implying the contradiction $L(v) \leq nr(y) < nr(u)$ (because of the path from v to x in T' followed by the edge xy). Hence each path connecting a vertex in T' with a vertex in S has to contain u, so that u is a cut point. □

Lemma 8.3.7. *Under the assumptions of Lemma 8.3.6, s is a cut point if and only if s is on at least two tree edges.*

Proof. First, let s be a cut point and $V_1 \dot\cup \ldots \dot\cup V_k$ (with $k \geq 2$) a partition of $V \setminus s$ for which all paths connecting two vertices in distinct components of the partition have to pass through s. Moreover, let $e = sv$ be the first (tree) edge traversed during the DFS; say $v \in V_1$. Then no vertex outside V_1 is accessible from v in T, so that s has to be incident with at least one further tree edge.

Conversely, let sv and sw be tree edges, and let T' be the part of T which has root v. By Lemma 8.2.4, there are no edges connecting a vertex of T' to a vertex in $V \setminus (T' \cup \{s\})$. As the set $V \setminus (T' \cup \{s\})$ is nonempty by hypothesis (it contains w), s is a cut point. □

Obviously, the only cut point in Example 8.3.5 is the vertex 5, in agreement with Lemmas 8.3.6 and 8.3.7. We now want to design a variant of Algorithm

8.2.1 which also computes the function L and determines both the cut points and the blocks of G. Let us first consider how L could be calculated. We set $L(v) := nr(v)$ when v is reached for the first time. If v is a leaf of T, the definition of L implies

$$L(v) \;=\; \min\{nr(u)\colon\; u = v \text{ or } vu \text{ is a back edge in } G\}\,. \tag{8.2}$$

Thus we replace $L(v)$ by $\min\{L(v), nr(u)\}$ as soon as the algorithm uses a back edge vu – that is, as soon as $nr(u) \neq 0$ during the examination of vu in step (6) or (7) of the DFS. When the algorithm backtracks from v to $p(v)$ in step (9), all back edges have been examined, so that $L(v)$ has obtained its correct value (8.2). Similarly, if v is not a leaf,

$$L(v) \;=\; \min\left(\{nr(u)\colon\; u = v \text{ or } vu \text{ a back edge }\} \cup \{L(u)\colon\; vu \in T\}\right); \tag{8.3}$$

in this case, we have to replace $L(v)$ by $\min\{L(v), L(u)\}$ as soon as a tree edge vu is used for the second time, namely from u to v. Then the examination of u is finished and $L(u)$ has its correct value, as may be shown by induction.

Now we are in a position to state the algorithm of Tarjan [Tar72]. It will be convenient to use a *stack* S for determining the blocks.[2] The reader should note that Algorithm 8.3.8 has precisely the same structure as the DFS in Algorithm 8.2.1 and therefore also the same complexity $O(|E|)$.

Algorithm 8.3.8. Let G be a connected graph and s a vertex of G. The algorithm determines the set C of cut points of G and the blocks of G and, thus, the number k of blocks of G).

Procedure BLOCKCUT$(G, s; C, k)$

(1) **for** $v \in V$ **do** $nr(v) \leftarrow 0$; $p(v) \leftarrow 0$ **od**
(2) **for** $e \in E$ **do** $u(e) \leftarrow$ false **od**
(3) $i \leftarrow 1$; $v \leftarrow s$; $nr(s) \leftarrow 1$; $C \leftarrow \emptyset$; $k \leftarrow 0$; $L(s) \leftarrow 1$;
(4) create a stack S with single element s;
(5) **repeat**
(6) **while** there exists $w \in A_v$ with $u(vw) =$ false **do**
(7) choose some $w \in A_v$ with $u(vw) =$ false; $u(vw) \leftarrow$ true;
(8) **if** $nr(w) = 0$
(9) **then** $p(w) \leftarrow v$; $i \leftarrow i + 1$; $nr(w) \leftarrow i$;
$L(w) \leftarrow i$; append w to S; $v \leftarrow w$
(10) **else** $L(v) \leftarrow \min\{L(v), nr(w)\}$
(11) **fi**
(12) **od**

[2] Recall that a stack is a list where elements are appended at the end and removed at the end as well (*last in – first out*), in contrast to a *queue* where elements are appended at the end, but removed at the beginning (*first in – first out*). For a more detailed discussion of these data structures (as well as for possible implementations), we refer to [AhHU74, AhHU83] or to [CoLR90].

(13) **if** $p(v) \neq s$
(14) **then if** $L(v) < nr(p(v))$
(15) **then** $L(p(v)) \leftarrow \min\{L(p(v)), L(v)\}$
(16) **else** $C \leftarrow C \cup \{p(v)\}$; $k \leftarrow k+1$;
(17) create a list B_k containing all vertices of S up to v (including v) and remove these vertices from S; append $p(v)$ to B_k
(18) **fi**
(19) **else if** there exists $w \in A_s$ with $u(sw) =$ false **then** $C \leftarrow C \cup \{s\}$ **fi**
(20) $k \leftarrow k+1$; create a list B_k containing all vertices of S up to v (including v) and remove these vertices from S; append s to B_k
(21) **fi**
(22) $v \leftarrow p(v)$
(23) **until** $p(v) = 0$ **and** $u(vw) =$ true for all $w \in A_v$

Theorem 8.3.9. *Algorithm 8.3.8 determines the cut points and the blocks of a connected graph G with complexity $O(|E|)$.*

Proof. As in the original DFS, each edge is used exactly once in each direction (see Theorem 8.2.2); moreover, for each edge a constant number of steps is executed. Hence Algorithm 8.3.8 has complexity $O(|E|)$. The considerations above show that $L(v)$ has the correct value given in (8.2) or (8.3), as soon as the algorithm has finished examining v (because the condition in step (6) is no longer satisfied). A formal proof of this fact may be given using induction on $nr(v)$ (in decreasing order). Note that the edge vw chosen in step (7) is a back edge if and only if $nr(w) \neq 0$ holds, and that the tree edge $p(v)v$ is used in step (15) for updating the value of $L(p(v))$ after $L(v)$ has been determined (unless this updating is redundant because of $L(v) \geq nr(p(v)) \geq L(p(v))$).

It remains to show that the cut points and the blocks are determined correctly. After the algorithm has finished examining the vertex v (according to the condition in (6)), it is checked in (14) or (19) whether $p(v)$ is a cut point. First suppose $p(v) \neq s$. If the condition in (14) is satisfied, the (correct) value of $L(v)$ is used to update $L(p(v))$ (as explained above); otherwise, $p(v)$ is a cut point by Lemma 8.3.6. In this case, $p(v)$ is added to the set C of cut points in step (16). The vertices in S up to v (including v) are descendants of v in T, where T is the arborescence determined by the DFS. Now these vertices are not necessarily all the descendants of v: it is possible that descendants of cut points were removed earlier; there might have been cut points among these vertices. However, it can be shown by induction that no proper descendants of such cut points are contained in S any more. (The induction basis – for leaves of T – is clear.) Therefore, the set B_k in step (17) is indeed a block.

Next, suppose $p(v) = s$. If the condition in step (19) holds, s is a cut point by Lemma 8.3.7. It can be shown as above that B_k is a block of G. In particular, s is added to C if and only if not all the edges incident with s were treated yet when $p(v) = s$ occurred for the first time.

In both cases, v is replaced at this point by its predecessor $p(v)$. By Lemmas 8.3.6 and 8.3.7, all cut points have been found when the algorithm terminates (after all edges have been used in both directions). □

Exercise 8.3.10. Execute Algorithm 8.3.8 for the graph displayed in Figure 8.5. If there are choices to be made, proceed in alphabetical order.

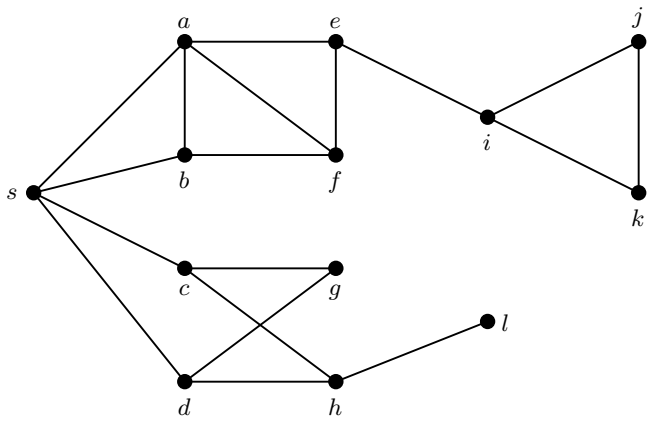


Fig. 8.5. A connected graph G

8.4 Depth first search for digraphs

In this section we discuss how the DFS given in Algorithm 8.2.1 should be performed for a digraph G. For this purpose, all edges vw are to be interpreted as directed from v to w (for example, in step (5) of the algorithm). In every other respect, the algorithm is executed as before, so that the only difference compared to the undirected case is that edges with $u(e) = \text{false}$ may be used in one direction only, namely as forward edges.

Even if G is connected, we will in general not reach all vertices of G. In fact, DFS$(G, s; nr, p)$ reaches exactly those vertices which are accessible from s by a directed path. We shall often assume that these are all the vertices of G; otherwise, DFS may be executed again for a vertex s' not accessible from s, etc. Basically, Theorem 8.2.2 remains valid: all those edges whose start vertex is accessible from s are used exactly once in each direction. Edges of the form $p(v)v$ are again called *tree edges*. If all vertices of G are accessible from s, the tree edges form a spanning arborescence of G with root s, as in Theorem 8.2.3; in the general case, we will obtain a directed spanning forest. All proofs proceed in complete analogy to the corresponding proofs for the undirected case given in Section 8.2, and will therefore be omitted. The following theorem summarizes these considerations.

Theorem 8.4.1. *Let G be a digraph and s a vertex of G. Moreover, let S be the set of vertices of G which are accessible from s. Then* $\mathrm{DFS}(G, s; nr, p)$ *reaches all the vertices of S and no other vertices (that is, $nr(v) \neq 0$ if and only if $v \in S \setminus \{s\}$); moreover, the tree edges $p(v)v$ form a spanning arborescence on S. The complexity of the algorithm is $O(|E|)$.* □

In the undirected case, there were only tree edges and back edges (see Lemma 8.2.4). For a digraph, we have to distinguish three kinds of edges beside the tree edges:

(1) *Forward edges*: these are edges of the form $e = vu$ such that u is a descendant of v, but not $v = p(u)$. In this case, we have $nr(u) > nr(v)$.
(2) *Back edges*: these are edges of the form $e = vu$ such that u is an ancestor of v; here, $nr(u) < nr(v)$.
(3) *Cross edges*: these are edges of the form $e = vu$ such that u is neither an ancestor nor a descendant of v. In particular, each edge connecting two distinct directed trees (if not all vertices of G are accessible from s) is a cross edge. Cross edges may also exist within a single directed tree; in that case, we have $nr(u) < nr(v)$.

Example 8.4.2. Let G be the digraph shown in Figure 8.6. Then a call of $\mathrm{DFS}(G, a; nr, p)$ followed by $\mathrm{DFS}(G, f; nr, p)$ yields the result drawn in Figure 8.7, where choices are made in alphabetical order (as usual). Tree edges are drawn bold, cross edges broken, and all other edges are in normal print. The only back edge is eb.

Exercise 8.4.3. Consider a digraph $G = (V, E)$. Let u and v be two vertices in a tree found by a call of $\mathrm{DFS}(G, s; nr, p)$, and assume $nr(u) > nr(v)$ and $e = vu \in E$. Show that e is indeed a forward edge.

Exercise 8.4.4. Let $G = (V, E)$ be a digraph and $T_1, \ldots, T_k$ a directed spanning forest partitioning V, found by repeated execution of DFS on G. Show that G is acyclic if and only if G does not contain any back edges.

8.5 Strongly connected digraphs

In analogy with the notion of blocks in a graph, the vertex set S of any maximal, strongly connected, induced subdigraph of a digraph G is called a *strong component* of G. Thus each vertex in S has to be accessible from each other vertex in S, and S is maximal with respect to this property. For example, the vertices b, c, d, e of the digraph shown in Figure 8.6 form a strong component; the remaining strong components of this digraph are singletons. Our first result collects some rather obvious equivalent conditions for strong connectedness; the proof is left to the reader.

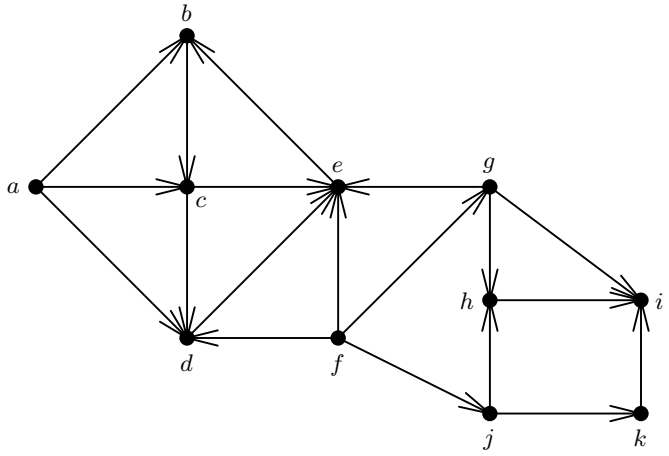


Fig. 8.6. A digraph G

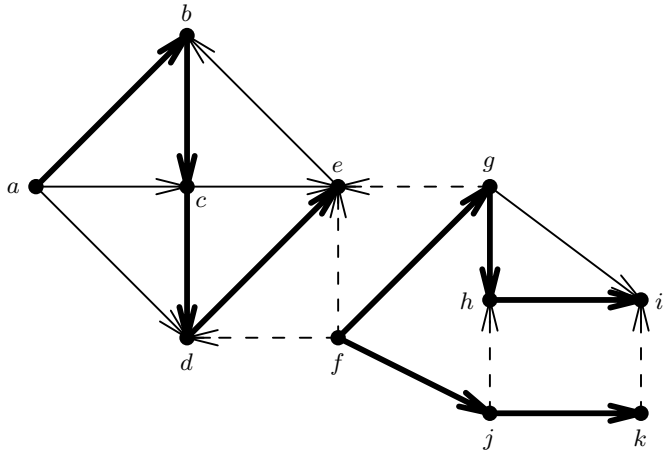


Fig. 8.7. The result of a DFS on G

Theorem 8.5.1. *Let G be a digraph with at least three vertices and with no isolated vertices. Then the following conditions are equivalent.*

(1) *G is strongly connected.*
(2) *For each pair of vertices of G, there exists a directed closed walk containing both of them.*
(3) *For each vertex v and for each edge e of G, there exists a directed closed walk containing both v and e.*
(4) *For each pair of edges of G, there exists a directed closed walk containing both of them.*

(5) *For each pair of vertices* (x, y) *and for each edge* e *of* G*, there exists a directed walk from* x *to* y *containing* e*.*
(6) *For each triple of vertices* (x, y, z)*, there exists a directed walk from* x *to* y *containing* z*.* □

Exercise 8.5.2. The reader will have noticed that the properties stated in Theorem 8.5.1 are similar to those given in Theorem 8.3.1 for 2-connected graphs; however, it uses walks instead of cycles or paths. Show by giving counterexamples that the analogous statement to (7) of Theorem 8.3.1 does not hold, and that the conditions in Theorem 8.5.1 do not hold if we replace closed walks and walks by cycles and paths, respectively.

Note that the underlying graph $|G|$ of a strongly connected digraph G is not necessarily 2-connected. On the other hand, a 2-connected graph cannot contain any bridges and is therefore orientable by Theorem 1.6.2.

Exercise 8.5.3. Let G be a connected digraph. Show that G is strongly connected if and only if every edge of G is contained in a directed cycle.

Our next aim is an algorithm for determining the strong components of a digraph G. The algorithm which we will present is taken from the book of Aho, Hopcroft and Ullman [AhHU83]; it consists of performing a DFS both for G and for the digraph having opposite orientation[3]. A further algorithm for this task was given by Tarjan [Tar72]; his algorithm requires to execute a DFS only once, but needs – similar to Algorithm 8.3.8 – the function $L(v)$. Tarjan's algorithm can also be found in the book [Eve79]. The basic concept of the algorithm we give below is considerably simpler; as both methods lead to the same complexity, we have chosen the simpler one. First we need to modify the DFS algorithm slightly: we will require a second labelling $Nr(v)$ of the vertices, according to the order in which the examination of the vertices is finished.

Algorithm 8.5.4. Let $G = (V, E)$ be a digraph and s a root of G.
Procedure DFSM$(G, s; nr, Nr, p)$

(1) **for** $v \in V$ **do** $nr(v) \leftarrow 0$; $Nr(v) \leftarrow 0$; $p(v) \leftarrow 0$ **od**
(2) **for** $e \in E$ **do** $u(e) \leftarrow$ false **od**
(3) $i \leftarrow 1$; $j \leftarrow 0$; $v \leftarrow s$; $nr(s) \leftarrow 1$; $Nr(s) \leftarrow |V|$;
(4) **repeat**
(5) **while** there exists $w \in A_v$ with $u(vw) =$ false **do**
(6) choose some $w \in A_v$ with $u(vw) =$ false; $u(vw) \leftarrow$ true;
(7) **if** $nr(w) = 0$ **then** $p(w) \leftarrow v$; $i \leftarrow i + 1$; $nr(w) \leftarrow i$; $v \leftarrow w$ **fi**
(8) **od**
(9) $j \leftarrow j + 1$; $Nr(v) \leftarrow j$; $v \leftarrow p(v)$
(10) **until** $v = s$ **and** $u(sw) =$ true for each $w \in A_s$.

[3] This means replacing each edge uv of G by vu.

Using this procedure, we can write down the algorithm of Aho, Hopcroft and Ullman for determining the strong components of G. We may assume that each vertex of G is accessible from s.

Algorithm 8.5.5. Let G be a digraph and s a root of G. The algorithm determines the strong components of G.

Procedure STRONGCOMP$(G, s; k)$

(1) DFSM$(G, s; nr, Nr, p)$; $k \leftarrow 0$;
(2) let H be the digraph with the opposite orientation of G;
(3) **repeat**
(4) choose the vertex r in H for which $Nr(r)$ is maximal;
(5) $k \leftarrow k + 1$; DFS$(H, r; nr', p')$; $C_k \leftarrow \{v \in H : nr'(v) \neq 0\}$;
(6) remove all vertices in C_k and all the edges incident with them;[4]
(7) **until** the vertex set of H is empty

Theorem 8.5.6. *Let G be a digraph with root s. Then Algorithm 8.5.5 calculates with complexity $O(|E|)$ the strong components $C_1, \ldots, C_k$ of G.*

Proof. The complexity of Algorithm 8.5.5 is clear. We have to show that the directed forest on the vertex sets $C_1, \ldots, C_k$ determined by the second DFS during the **repeat**-loop indeed consists of the strong components of G.

Thus let v and w be two vertices in the same strong component of G. Then there exist directed paths from v to w and from w to v in G, and hence also in H as well. We may suppose that v is reached before w during the DFS on H. Moreover, let T_i be the directed tree containing v, and x the root of T_i. As w is accessible from v in H and was not examined before, w has to be contained in T_i as well: Theorem 8.4.1 implies that w is reached during the execution of the DFS with root x.

Conversely, let v and w be two vertices contained in the same directed tree T_i (on C_i). Again, let x be the root of T_i; we may suppose $v \neq x$. As v is a descendant of x in T_i, there exists a directed path from x to v in H and hence a directed path from v to x in G. Now v was not yet examined when the DFS on H with root x began, so that $Nr(v) < Nr(x)$ because of (4). Thus the examination of v was finished earlier than the examination of x during the DFS on G; see step (9) in Algorithm 8.5.4. But as x is accessible from v in G, v cannot have been reached earlier than x during the DFS on G. This means that the entire examination of v was done during the examination of x, so that v has to be a descendant of x in the spanning tree T for G. Hence there also exists a directed path from x to v in G, and x and v are contained in the same strong component. Similarly, w has to be contained in the same strong component. □

[4] Note that the resulting digraph is still denoted by H.

Example 8.5.7. We apply Algorithm 8.5.5 to the digraph G of Figure 8.6. As a is not a root of G, we have to modify the algorithm slightly or apply it twice (from a and from f). Figure 8.8 shows the digraph H and the result of the DFS on G modified as in 8.5.4. All edges of H have orientation opposite to the orientation they have in G; the numbers given are the values $Nr(v)$ calculated by calls of DFSM$(G, a; nr, p)$ and DFSM$(G, f; nr, p)$. The cross edges connecting the two directed trees are omitted. In Figure 8.9, the strong components as determined by Algorithm 8.5.5 are drawn; to make the figures simpler, we leave out all the edges connecting distinct strong components. Note that a DFS on H using a different order of the start vertices – beginning at e, for example – would yield an incorrect result.

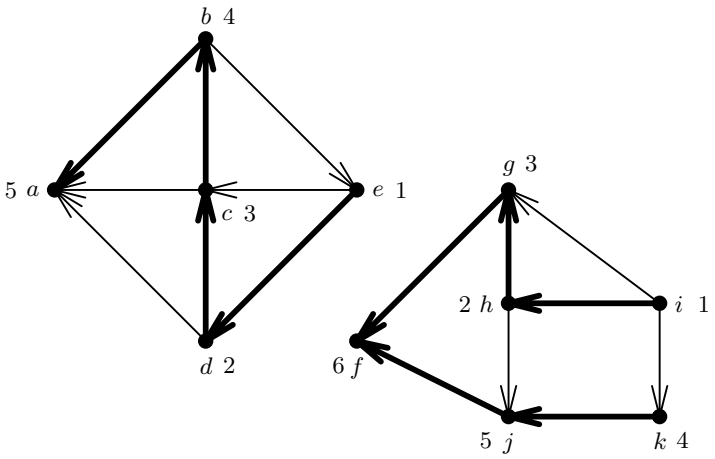


Fig. 8.8. Directed graph H with Nr-labels

Exercise 8.5.8. Determine the strong components of the digraph displayed in Figure 3.3.

Exercise 8.5.9. Let $C_1, \ldots, C_k$ be the strong components of a digraph G. We define a digraph G', the *condensation* of G, as follows: the vertices of G' are $C_1, \ldots, C_k$; C_iC_j is an edge of G' if and only if there exists an edge uv in G with $u \in C_i$ and $v \in C_j$. Show that G' is acyclic and determine G' for the digraph of Figure 3.3; compare Exercise 8.5.8.

Exercise 8.5.10. Give a definition of the term *strongly k-connected* for digraphs and investigate whether the main results of Section 8.1 carry over.

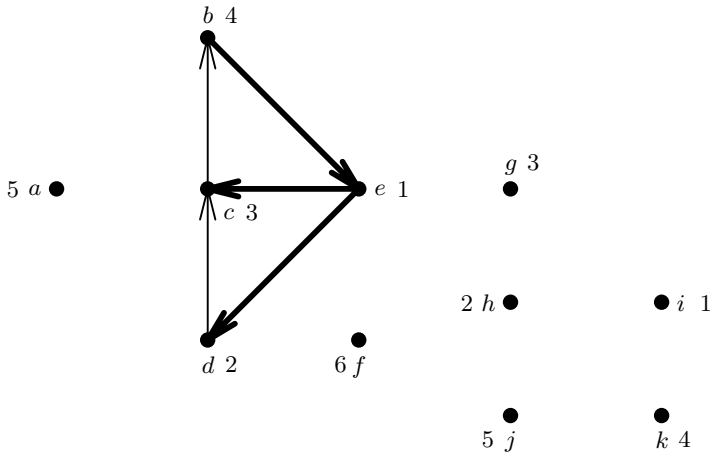


Fig. 8.9. Result of the DFS on H and strong components

8.6 Edge connectivity

We finish this chapter by considering notions of connectivity which arise from replacing vertex separators and vertex disjoint paths by edge separators and edge disjoint paths, respectively. Let G be a graph or a digraph, and let u and v be two distinct vertices of G. By $\lambda(u, v)$ we denote the minimal cardinality of an edge separator for u and v. By Theorem 7.1.1, $\lambda(u, v)$ is also the maximal number of edge disjoint paths (directed if G is a digraph) from u to v. The *edge connectivity* $\lambda(G)$ is defined as

$$\lambda(G) \;=\; \min\{\lambda(u, v) : u, v \in V\}.$$

G is called *m-fold edge connected*[5] if $\lambda(G) \geq m$. Moreover, $\delta(G)$ denotes the minimal degree of a vertex of G if G is a graph, and the minimum of all $d_{\text{in}}(v)$ and all $d_{\text{out}}(v)$ if G is a digraph. We have the following simple result; see [Whi32a].

Theorem 8.6.1. *Let G be a graph or a digraph. Then $\kappa(G) \leq \lambda(G) \leq \delta(G)$.*

Proof. We consider only the undirected case; the directed case is similar. Let v be a vertex with $\deg v = \delta(G)$. Removing all edges incident with v obviously yields a disconnected graph, so that $\lambda(G) \leq \delta(G)$. If $\lambda(G) = 1$, G contains a bridge $e = uv$. Then G cannot be 2-connected, because removing u from G yields either a K_1 or a disconnected graph. If $\lambda(G) = k \geq 2$, removing $k - 1$ edges $e_2, \ldots, e_k$ of an edge separator from G results in a graph H containing a bridge $e_1 = uv$. Therefore, if we remove from G one of the end vertices of each

[5] Some authors use the terms *line connectivity* and *line connected* instead.

of the e_i distinct from u and v (for $i = 2, \ldots, k$), we get either a disconnected graph or a graph where e_1 is a bridge (so that removing u makes the graph disconnected). In either case, $\kappa(G) \leq k = \lambda(G)$. □

The graph in Figure 8.10 shows that the inequalities of Theorem 8.6.1 may be strict. This graph arises from a considerably more general construction [ChHa68]:

Exercise 8.6.2. Fix integers k, d, and m with $0 < k \leq m \leq d$. Find a graph with $\kappa(G) = k$, $\lambda(G) = m$, and $\delta(G) = d$. Hint: Distinguish the cases $k = d$ and $k \neq d$.

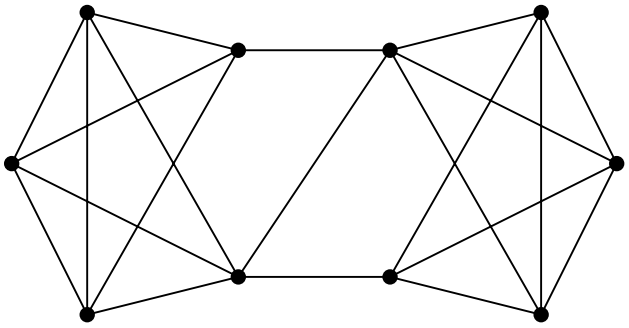


Fig. 8.10. A graph with $\kappa = 2$, $\lambda = 3$, and $\delta = 4$

Exercise 8.6.3. Let G be a graph with n vertices. Show $\lambda(G) = \delta(G)$ provided that $\delta(G) \geq n/2$. Is this bound tight? See [Cha66].

The following lemma, due to Schnorr [Schn79], is useful for computing $\lambda(G)$ because it allows one to determine $\min\{\lambda(u, v) \colon u, v \in V\}$ by calculating $\lambda(u, v)$ for only a few pairs (u, v).

Lemma 8.6.4. *Let G be a graph or a digraph with vertex set $V = \{v_1, \ldots, v_n\}$. Then, with $v_{n+1} = v_1$:*

$$\lambda(G) \;=\; \min\{\lambda(v_i, v_{i+1}) \colon i = 1, \ldots, n\}.$$

Proof. Let u and v be vertices of G satisfying $\lambda(G) = \lambda(u, v)$, and let T be an edge separator of cardinality $\lambda(u, v)$ for u and v. Denote the set of all vertices w for which there is a path (directed if G is a digraph) from u to w not containing any edge of T by X; similarly, Y denotes the set of all vertices w for which each path (directed if G is a digraph) from u to w contains some edge from T. Then (X, Y) is a cut of G, with $u \in X$ and $v \in Y$. Now T is an edge separator for x and y for each pair of vertices with $x \in X$ and $y \in Y$:

otherwise, there would be a path from u to y not containing any edges from T. Hence $|T| = \lambda(G) \leq \lambda(x, y) \leq |T|$; that is, $\lambda(x, y) = \lambda(G)$. Obviously, there has to exist an index i such that $v_i \in X$ and $v_{i+1} \in Y$; then $\lambda(G) = \lambda(v_i, v_{i+1})$ for this i. □

The reader is invited to explore why an analogous argument for vertex separators does not work. By Corollary 7.1.2, each $\lambda(u, v)$ can be determined with complexity $O(|V|^{2/3}|E|)$. Hence Lemma 8.6.4 immediately yields the following result.

Theorem 8.6.5. *The edge connectivity $\lambda(G)$ of a graph or a digraph G can be determined with complexity $O(|V|^{5/3}|E|)$.* □

With a little more effort, one may improve the complexity bound of Theorem 8.6.5 to $O(|V||E|)$; see [Mat87] and [MaSc89]. Finally, we mention two interesting results concerning edge connectivity; proofs can be found in [Eve79] or in the original papers [Edm73] and [EvGT77].

Result 8.6.6. *Let G be a digraph and u a vertex of G. Then there exist k edge disjoint directed spanning trees of G with common root u, where $k = \min\{\lambda(u, v) : v \neq u\}$.* □

Result 8.6.7. *Let G be a digraph with $\lambda(G) \geq k$. For each pair of vertices (u, v) and for every m with $0 \leq m \leq k$, there are m directed paths from u to v and $k - m$ directed paths from v to u, all of which are edge disjoint.* □

We refer to [Bol78] for a treatment of extremal cases; a typical problem of this sort is the determination of the structure of 2-connected graphs for which removing any edge destroys the 2-connectedness.

9

Colorings

More delicate than the historians'
are the map-makers' colors.
Elizabeth Bishop

This chapter treats a subject occurring quite often in graph theory: colorings. We shall prove the two fundamental major results in this area, namely the theorems of Brooks on vertex colorings and the theorem of Vizing on edge colorings. As an aside, we explain the relationship between colorings and partial orderings, and briefly discuss perfect graphs. Moreover, we consider edge colorings of Cayley graphs; these are graphs which are defined using groups. Finally, we turn to map colorings: we shall prove Heawood's five color theorem and report on the famous four color theorem. Our discussion barely scratches the surface of the vast area; for a detailed study of coloring problems we refer the reader to the monograph [JeTo95].

9.1 Vertex colorings

In this section we prove some basic results about the chromatic number $\chi(G)$ of a graph G. We need some definitions: a *coloring* of a graph $G = (V, E)$ assigns a *color* to each of its vertices so that adjacent vertices always have different colors.[1] More formally, we have a map $c : V \to C$ into some set C which we interpret as the set of *colors*, and we require $c(v) \neq c(w)$ for every edge $vw \in E$. The *chromatic number* $\chi(G)$ is the minimal number of colors needed in a coloring of G.

To obtain a feeling for these concepts, let us give a simple example and an illuminating exercise:

Example 9.1.1. Obviously, a graph G has chromatic number 2 if and only if its vertex set can be partitioned into two subsets S and T so that no edge has both its end vertices in one of S or T. Thus the graphs G with $\chi(G) = 2$ are precisely the (nonempty) bipartite graphs.

[1] Sometimes an *arbitrary* assignment of colors to the vertices is called a *coloring*. Then colorings for which adjacent vertices always have different colors are called *admissible colorings*.

Exercise 9.1.2. Show that the icosahedral graph G (see Figure 9.1) has chromatic number $\chi(G) = 4$ by

- proving that G cannot be colored with three colors, and
- displaying a coloring with four colors.

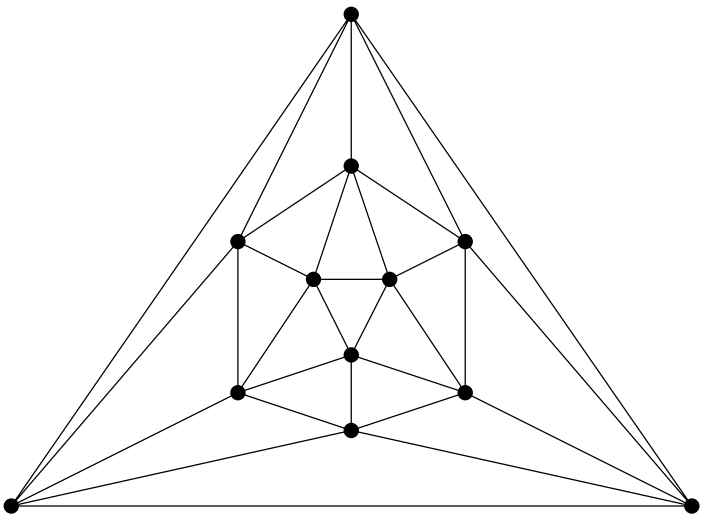


Fig. 9.1. The icosahedral graph

How could we actually color a given graph? An obvious approach is provided by the following greedy-type algorithm. We may assume that the colors used are the positive integers $1, 2, \ldots$. Given any ordering $v_1, \ldots, v_n$ of the vertices of G, we color the vertices one by one, where we always use the smallest available color. Here is a formal version of this idea:

Algorithm 9.1.3. Let $G = (V, E)$ be a graph given by adjacency lists A_i, and let $\mathbf{v} = (v_1, \ldots, v_n)$ be an ordering of the vertices of G. The algorithm constructs a coloring c of G with colors $1, 2, \ldots$.

Procedure COLOR$(G, \mathbf{v}; f)$.

(1) $c(v_1) \leftarrow 1$;
(2) **for** $i = 2$ **to** n **do**
(3) $\quad c(v_i) \leftarrow \min \{j : c(v_h) \neq j$ for all $h = 1, \ldots, i-1$ with $h \in A_i \}$
(4) **od**

The quality of the preceding procedure will depend on the ordering of the vertices. Nevertheless, even an arbitrary ordering leads to a pretty good bound:

Lemma 9.1.4. *Let G be a graph. Then $\chi(G) \leq \Delta(G)+1$, where $\Delta(G)$ denotes the maximal degree of a vertex of G.*

Proof. Let $v_1, \ldots, v_n$ be an arbitrary ordering of the vertices of G. Now v_i has at most $\Delta(G)$ adjacent predecessors when it is colored in step (3) of Algorithm 9.1.3; hence there are at most $\Delta(G)$ colors which are not admissible for coloring v_i so that $c(v_i) \leq \Delta(G) + 1$. □

Example 9.1.5. There are graphs for which equality holds in Lemma 9.1.4, namely complete graphs and cycles of odd length: $\chi(K_n) = n = \Delta(K_n) + 1$ and $\chi(C_{2n+1}) = 3 = \Delta(C_{2n+1})+1$. On the other hand, $\chi(C_{2n}) = 2 = \Delta(C_{2n})$.

Using an appropriate ordering of the vertices, we can prove the following stronger bound; its main interest lies in the subsequent corollary.

Lemma 9.1.6. *Let $G = (V, E)$ be a graph. Then:*

$$\chi(G) \;\leq\; 1 + \max\{\delta(H) \colon H \text{ is an induced subgraph of } G\},$$

where $\delta(H)$ denotes the minimal degree of a vertex of H.

Proof. Denote the maximum defined above by k; by hypothesis, $k \geq \delta(G)$. Thus we may choose a vertex v_n with $\deg v_n \leq k$. Now look at the induced subgraph $H = H_{n-1} := G \setminus v_n$ and choose a vertex v_{n-1} having at most degree k in H. We continue in this manner until we get a graph H_1 consisting of only one vertex, namely v_1. This determines an ordering $v_1, \ldots, v_n$ of V; we apply Algorithm 9.1.3 with respect to this ordering. Then each vertex v_i has at most k adjacent predecessors, because $\deg v_i \leq k$ holds in H_i. Thus Algorithm 9.1.3 needs at most $k + 1$ colors for coloring G. □

Corollary 9.1.7. *Let G be a connected graph, and assume that G is not regular. Then $\chi(G) \leq \Delta(G)$.*

Proof. Write $k = \Delta(G)$ and suppose $\chi(G) > k$. Then G has to have an induced subgraph H with $\delta(H) = k$, by Lemma 9.1.6. As k is the maximal degree in G, H must be a k-regular subgraph which is not connected to any vertex outside of H. Since G is connected, we conclude $G = H$, so that G has to be k-regular. □

We now come to the major result of this section, namely the theorem of Brooks [Bro41]: with the obvious exceptions mentioned in Example 9.1.5, $\chi(G) \leq \Delta(G)$.

Theorem 9.1.8 (Brooks' theorem). *Let G be a connected graph which is neither complete nor a cycle of odd length. Then $\chi(G) \leq \Delta(G)$.*

Proof. In view of Corollary 9.1.7, we may assume that G is regular of degree $k = \Delta(G)$. Now $k = 2$ means that G is a cycle, and we know the chromatic number of a cycle from Example 9.1.5. Hence we may assume $k \geq 3$. First suppose that G is not 2-connected. Then there exists a vertex v such that $G \setminus v$ is not connected; see Definition 7.1.6. Let $V_1, \ldots, V_l$ be the connected components of $G \setminus v$. Using induction on the number of vertices, we may assume that the subgraph of G induced on $V_i \cup \{v\}$ can be colored with k colors. Then G can obviously be colored using k colors as well. Thus we may from now on assume that G is 2-connected.

The assertion follows easily if we can find three vertices v_1, v_2, and v_n such that the graph $H = G \setminus \{v_1, v_2\}$ is connected, and G contains the edges v_1v_n and v_2v_n, but no edge joining v_1 to v_2. Suppose we have already found such vertices; then we may order the remaining vertices as follows: for $i = n-1, \ldots, 3$, we choose (in decreasing order, beginning with $i = n-1$) a vertex $v_i \in V \setminus \{v_1, v_2, v_{i+1}, \ldots, v_n\}$ adjacent to at least one of the vertices $v_{i+1}, \ldots, v_n$; note that this is indeed possible, since H is connected. Now we apply Algorithm 9.1.3 using this ordering of the vertices. Then we first get $c(v_1) = c(v_2) = 1$, as v_1 and v_2 are not adjacent. Furthermore, each vertex v_i with $3 \leq i \leq n-1$ has at most $k-1$ adjacent predecessors: v_i is adjacent to at least one vertex v_j with $j > i$. Finally, v_n is adjacent to the vertices v_1 and v_2 which have the same color. Therefore, the algorithm needs at most k colors.

It remains to show that G contains vertices v_1, v_2, and v_n satisfying the above conditions. First suppose that G is even 3-connected. Then we may choose an arbitrary vertex for v_n. Note that the set $\Gamma(v_n)$ of vertices adjacent to v_n has to contain two non-adjacent vertices v_1 and v_2. (Otherwise the $k+1$ vertices in $\Gamma(v_n) \cup \{v_n\}$ form a complete graph K_{k+1}; because of the connectedness and the k-regularity of G, this graph would have to be G itself, which contradicts the hypothesis of the theorem.) As G is 3-connected, H must still be connected.

Finally, we turn to the case where G is 2-connected but not 3-connected. Here we may choose a vertex separator $\{v, v_n\}$. Let $V_1, \ldots V_k$ be the connected components of $G \setminus \{v, v_n\}$ and put $G_i = G|(V_i \cup \{v, v_n\})$. Then the graphs G_i are connected; moreover, v_n has to have some neighbor $\neq v$ in each of the G_i, as otherwise $G \setminus v$ would not be connected. Now we choose two neighbors $v_1 \in G_1$ and $v_2 \in G_2$ of v_n such that $v_1, v_2 \neq v$. Then v_1 and v_2 are not adjacent and $H := G \setminus \{v_1, v_2\}$ is still connected which is seen as follows. Let x be any vertex of H. It suffices to show that v is still accessible from x in H. Now G is 2-connected, so that there are two vertex disjoint paths from x to v in G by Theorem 7.1.4; obviously, H contains at least one of these paths. This shows that H is connected and concludes the proof of the theorem. □

Let us make some remarks about the theorem of Brooks. The bound $\chi(G) \leq \Delta(G)$ may be arbitrarily bad: for bipartite graphs, $\chi(G) = 2$; and $\Delta(G)$ can take any value ≥ 2. In general, determining $\chi(G)$ is an NP-hard

problem; see [Kar72]. If P $\neq$ NP holds, then there is not even a polynomial algorithm producing an approximate solution which always needs less than $2\chi(G)$ colors; see [GaJo76]. However, there is an algorithm of complexity $O(|V| + |E| \log k)$ which, with probability almost 1, colors any given k-colorable graph with k colors [Tur88].

Exercise 9.1.9. Determine the chromatic number of the Petersen graph, using a theoretical argument. Also produce an explicit coloring confirming your result.

In the context of colorings, there are two important conjectures concerning the structure of k-colorable graphs. The first of these is one of the most famous open problems in graph theory: *Hadwiger's conjecture* [Had43] asserts that G contains a subgraph contractible to K_n provided that $\chi(G) \geq n$. This conjecture has been proved for $n \leq 6$; for $n \leq 4$, the proof can be found in Theorem 5.13 of [Aig84]. By a result of Wagner [Wag60], the case $n = 5$ is equivalent to the four color theorem (see Section 9.5) and is therefore proved as well. Finally, the case $n = 6$ was established by Robertson, Seymour and Thomas [RoST93]. The general case remains open; see [Tof96] for a survey. The second important conjecture sounds similar but is in fact stronger. *Hajós' conjecture* [Haj61] asserts that every graph with $\chi(G) \geq n$ contains a subdivision of K_n. However, Catlin [Cat79] found a counterexample for $n = 8$ (and hence for all $n \geq 8$), so that this conjecture is false in general. (Note that a subdivision of K_n can be contracted to K_n.) Hajós' conjecture is true for $n \leq 4$; for $n = 5, 6, 7$ it remains open; see [Aig84].

For more profound studies of the chromatic number, algebraic tools – in particular, the *chromatic polynomial* – are needed. This is one of the central topics in algebraic combinatorics; we refer the interested reader to [God93] or [Tut84].

9.2 Comparability graphs and interval graphs

In this section, we will apply the results of Section 7.5 to study the chromatic number and related parameters for two particularly interesting classes of graphs. We have already associated an acyclic directed graph with a given partial ordering in Section 2.6; now we also associate an undirected graph with a given poset. Thus let $(M, \preceq)$ be a partially ordered set. We define a graph G with vertex set M by choosing all those sets $\{x, y\}$ (where $x \neq y$) as edges of G for which x and y are *comparable*: $x \prec y$ or $y \prec x$ holds. Any such graph G is called a *comparability graph*. Note that a graph G is a comparability graph if and only if it has a transitive orientation. It it possible to check with complexity $O(|V|^{5/2})$ whether some given graph belongs to this class, and such a graph can be oriented with complexity $O(|V|^2)$; see [Spi85].

We require two more definitions. The maximal cardinality of a clique in a graph $G = (V, E)$ is called the *clique number* and will be denoted by $\omega(G)$. The *clique partition number* $\theta(G)$ is the minimal number of cliques in a partition of V into cliques. Let us note some simple relations between the independence number α defined Section 7.5 and the parameters ω, θ, and χ of a graph G and its complementary graph $\overline{G}$.

Lemma 9.2.1. *Let $G = (V, E)$ be a graph. Then:*

$$\chi(G) \geq \omega(G); \quad \alpha(G) \leq \theta(G); \quad \alpha(G) = \omega(\overline{G}); \quad \theta(G) = \chi(\overline{G}).$$

Proof. The first inequality holds, since the vertices of a clique obviously have to have distinct colors. Similarly, the vertices of an independent set have to be in distinct cliques, which yields the second inequality. Finally, independent sets in G are precisely the cliques in $\overline{G}$, and a coloring of $\overline{G}$ is equivalent to a partition of V into independent sets. □

Theorem 9.2.2. *Let G be a comparability graph or the complement of such a graph. Then $\alpha(G) = \theta(G)$ and $\omega(G) = \chi(G)$.*

Proof. Let G be a comparability graph. Then the cliques in G are precisely the chains of the corresponding partially ordered set $(M, \preceq)$, and the independent sets in G are the antichains of $(M, \preceq)$. Hence Theorem 7.5.3 implies $\alpha(G) = \theta(G)$, and Exercise 7.5.9 yields $\omega(G) = \chi(G)$. The assertion for $\overline{G}$ then follows using Lemma 9.2.1. □

Let us have a closer look at the complements of comparability graphs. Let $M_1, \ldots, M_n$ be intervals of real numbers, and G the graph on $\{1, \ldots, n\}$ whose edges are precisely the sets $\{i, j\}$ with $M_i \cap M_j \neq \emptyset$. Such a graph is called an *interval graph*.

Lemma 9.2.3. *Every interval graph is the complement of a comparability graph.*

Proof. Let $M_1, \ldots, M_n$ be intervals of real numbers and G the corresponding interval graph. We define a partial ordering $\preceq$ on $\{1, \ldots, n\}$ by

$$i \prec j \quad :\Longleftrightarrow \quad x < y \text{ for all } x \in M_i \text{ and all } y \in M_j.$$

The reader should check that this indeed yields a partial ordering. Obviously, $\{i, j\}$ is an edge in the comparability graph corresponding to $\preceq$ if and only if $M_i \cap M_j = \emptyset$; that is, iff $\{i, j\}$ is not an edge of G. □

Exercise 9.2.4. Show that every interval graph is *chordal* (or *triangulated*): each cycle of length at least 4 has a *chord*, that is, an edge of G which connects two non-consecutive vertices of the cycle.

Conversely, every chordal graph whose complement is a comparability graph has to be an interval graph; see [GiHo64]. Also, a graph G is a comparability graph if and only if every closed trail (not necessarily a cycle) $(v_0, v_1, \ldots, v_{2n}, v_{2n+1} = v_0)$ of odd length has a chord of the form $v_i v_{i+2}$; see [Gho62] and [GiHo64]. Proofs for these results can also be found in [Ber73], Chapter 16; further characterizations are given in [Gal67]. The paper [Fis85] contains more about interval graphs; algorithms for recognizing interval graphs are in [BoLu76] and [KoMo89].

Corollary 9.2.5. *Let G be an interval graph or the complement of such a graph. Then $\alpha(G) = \theta(G)$ and $\omega(G) = \chi(G)$.* □

Example 9.2.6. One often encounters interval graphs in practical applications, where the intervals are time intervals needed for performing certain tasks. A coloring of such a graph with as few colors as possible then corresponds to an optimal assignment of the given set of jobs to the minimum possible number of workers (or teams). As a concrete example, we mention scheduling flights to available planes and/or crews. Of course, practical applications usually involve many additional constraints. In particular, the scheduling of flights is a very important but also highly complex problem; see, for instance, [Yu98]. Applications of interval graphs in biology are described in [MiRo84].

Note that a coloring of an interval graph G is given by a partition of the associated comparability graph – the complement of G – into as few cliques as possible; that is, using the theorem of Dilworth, by a partition of the corresponding partial ordering into chains (compare the proof of Lemma 9.2.3). As mentioned at the end of Section 7.5, such a partition can be determined using a flow network. An explicit algorithm avoiding the use of flows can be found in [FoFu62], §II.9.

Comparability graphs and interval graphs are special instances of a very important class of graphs which we can only discuss briefly. A graph G is called *perfect* if the condition $\alpha(H) = \theta(H)$ holds for every induced subgraph H of G. Equivalently, G is perfect if and only if every induced subgraph H satisfies $\omega(H) = \chi(H)$. The fact that these two conditions are equivalent was first conjectured by Berge [Ber61] and finally proved by Lovász [Lov72]; see also [Gas96] for a considerably simpler proof. Alternatively, using Lemma 9.2.1, the result can also be formulated as follows:

Result 9.2.7 (perfect graph theorem). *The complement of a perfect graph is likewise perfect.*

Obviously, an induced subgraph of a comparability graph is again a comparability graph, so that we can summarize the results of this section as follows.

Theorem 9.2.8. *Comparability graphs, interval graphs, and the complements of such graphs are perfect.* □

Exercise 9.2.9. Show that bipartite graphs are perfect.

More about perfect graphs can be found in [Ber73], Chapter 16; for example, it is shown that every chordal graph is perfect. A stronger conjecture also posed by Berge [Ber61] remained unsolved for about 40 years before it was turned into a theorem in 2002 by Chudnovsky, Robertson, Seymour and Thomas [ChRST02]; see also [ChRST03] for background and an outline of the proof.

Result 9.2.10 (strong perfect graph theorem). *A graph is perfect if and only if neither G nor $\overline{G}$ contain a cycle of odd length ≥ 5 as an induced subgraph.*

Note that determining α, θ, ω, and χ is an NP-hard problem for graphs in general. Hence it is quite likely that no good algorithm exists for this problem; see [GaJo79]. However, all these parameters can be found in polynomial time for perfect graphs; see [GrLS84]. Thus perfect graphs are particularly interesting from an algorithmic point of view as well. This result and further interesting papers can be found in [BeCh84] which is devoted entirely to perfect graphs; see also [Gol80] and Chapter 9 of [GrLS93].

9.3 Edge colorings

In this section we treat *edge colorings*; this means we assign a *color* to each edge so that any two edges having a vertex in common have distinct colors. The smallest possible number of colors needed for an edge coloring of a graph G is called the *chromatic index* or the *edge chromatic number*; it is denoted by $\chi'(G)$. Note $\chi'(G) = \chi(L(G))$, where $L(G)$ is the line graph of G. The counterpart of the theorem of Brooks about vertex colorings is the theorem of Vizing [Viz64]. There is a remarkable difference, though: while the bound for $\chi(G)$ in Brooks' theorem can be arbitrarily bad, the theorem of Vizing guarantees that $\chi'(G)$ can only take one of two possible values, namely either $\Delta(G)$ or $\Delta(G) + 1$.

Exercise 9.3.1. Prove $\chi'(G) = \Delta(G)$ for bipartite graphs G. Hint: Use Exercise 7.4.16.

Theorem 9.3.2 (Vizing's theorem). *Let G be a graph. Then either $\chi'(G) = \Delta(G)$ or $\chi'(G) = \Delta(G) + 1$.*

Proof. The inequality $\chi'(G) \geq \Delta(G)$ is obvious. Using induction on $m = |E|$, we prove $\chi'(G) \leq \Delta(G) + 1$. The induction basis $m = 1$ is trivial. Now choose any edge $e_1 = uv$ of G and assume that $G \setminus e_1$ has already been colored using $\Delta(G) + 1$ colors; we will use this coloring of $G \setminus e_1$ to construct a coloring of G. We need more notation. For any two colors α and β, let $G(\alpha, \beta)$ be the subgraph of G whose edges are precisely those edges of G which have color

α or β. Obviously, the connected components of $G(\alpha, \beta)$ are paths or cycles of even length whose edges are alternately colored with α and β. Note that interchanging the two colors α and β in some of the connected components of $G(\alpha, \beta)$ yields a valid coloring again.

As each vertex v has degree at most $\Delta(G)$, there is at least one color γ missing at v in the coloring of $G \setminus e_1$: none of the edges incident with v has been assigned the color γ. If the same color is missing at u and at v_1, we may assign this color to the edge e_1. Now suppose that this is not the case; say color α is missing at u, and color $\beta_1 \neq \alpha$ is missing at v_1. Also, we may assume that some edge incident with v_1 is colored with α, and some edge $e_2 = uv_2$ is colored with β_1. We change the given coloring as follows: we assign color β_1 to edge e_1 and leave edge e_2 without a color for the moment. If α is the color missing at v_2, we may color e_2 with α. So suppose that α is assigned to some edge incident with v_2. If u, v_1, and v_2 are not in the same connected component of $G(\alpha, \beta_1)$, we can exchange the colors α and β_1 in the component containing v_2, so that α is then missing at v_2 (and α is still missing at u); then we may assign α to e_2. Otherwise, u, v_1, and v_2 are contained in the same connected component of $G(\alpha, \beta_1)$, which means that there is a path from u to v_2 alternately colored with α and β_1. This path together with the (not yet colored) edge e_2 forms a cycle; see Figure 9.2.

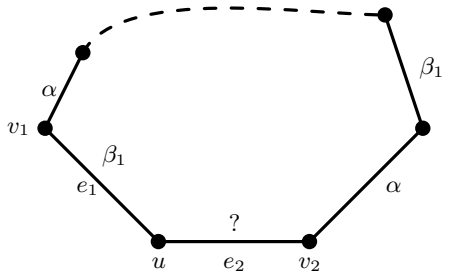


Fig. 9.2. A cycle in G

Now suppose that the color missing at v_2 is $\beta_2 \neq \beta_1$. We may also assume that this color occurs at u (for otherwise we might assign β_2 to e_2 and obtain the desired valid coloring); let $e_3 = uv_3$ be the edge colored with β_2. We change the given coloring as follows: we assign β_2 to e_2 and leave e_3 without a color for the moment. As before, we may assume that α occurs at v_3 and that u, v_2, and v_3 lie in the same connected component of $G(\alpha, \beta_2)$ (otherwise, it would be possible to find a color for e_2 and finish the coloring as above). Thus there is a path from u to v_3 colored alternately with α and β_2, and this path together with the (not yet colored) edge e_3 forms a cycle.

We have now found two *alternating cycles* as shown in Figure 9.3. We continue to change the coloring in the same manner, until we reach some

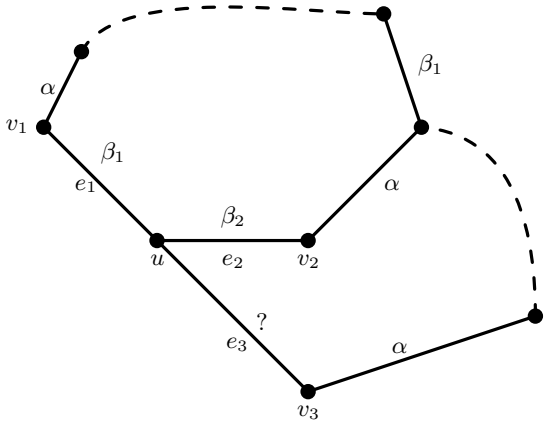


Fig. 9.3. Two cycles in G

vertex v_k adjacent to u for which the edge $e_k = uv_k$ is not yet colored and one of the following two cases occurs. Either some color $\beta_k \neq \beta_{k-1}$ is missing at v_k, and this color is also missing at u; in this case, e_k may be assigned this color. Or some color β_i with $i \leq k-2$ is missing at v_k. (Note that one of these alternatives has to occur at some point, because we can find at most $\deg u \leq \Delta(G)$ neighbors of u.) As before, u, v_i, and v_{i-1} are contained in the same connected component of $G(\alpha, \beta_i)$. This component is a path P from u to v_{i+1} alternately colored with α and β_i, and this path does not contain v_k, since β_i is missing at v_k. Thus the component C of $G(\alpha, \beta_i)$ containing v_k is disjoint from P; see Figure 9.4. Now we exchange the colors α and β_i in C; then we may assign α to e_k to finish our coloring. □

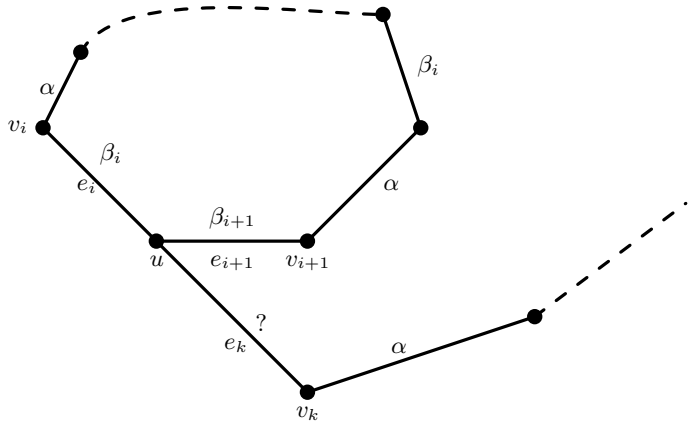


Fig. 9.4. Concluding the proof of Theorem 8.3.2

A short proof for a generalization of Vizing's theorem can be found in [BeFo91]; this also provides an alternative proof of Theorem 9.3.2.

As we saw in Exercise 9.3.1, bipartite graphs always have chromatic index $\chi'(G) = \Delta(G)$. As a consequence of Vizing's theorem, we can now also determine $\chi'(G)$ for all regular graphs.

Corollary 9.3.3. *Let G be a k-regular graph with n vertices. Then $\chi'(G) = k+1$ whenever n is odd. If n is even, $\chi'(G) = k$ if and only if G admits a 1-factorization.*

Proof. First let n be odd, say $n = 2m+1$. Then a given color can be assigned to at most m edges. As G contains $k(2m+1)/2 > mk$ edges, $\chi'(G) = k$ is impossible; hence $\chi'(G) = k+1$, by Theorem 9.3.2.

Now let n be even, say $n = 2m$. By a similar argument, $\chi'(G) = k$ if and only if each color is assigned to exactly m edges, which happens iff the color classes of edges form a 1-factorization. □

Exercise 9.3.4. Determine the chromatic index of the Petersen graph and display a corresponding edge coloring.

We refer to the monograph [Yap86] for an extensive discussion of edge colorings. Shannon [Sha49a] proved that multigraphs have chromatic index $\chi'(G) \leq 3\Delta(G)/2$. Some further results in this direction can be found for example in [And77] and [HiJa87] who proved that $\chi'(G) \leq \Delta(G) + 1$ still holds if all edges occurring more than once in G form a matching.

Even though Vizing's theorem restricts the chromatic index of a graph G to only two possible values, determining $\chi'(G)$ is an NP-hard problem. Holyer [Hol81] proved that this continues to hold when G is restricted to the class of 3-regular graphs. There are fast algorithms with complexity $O(|E|\log|E|)$ for the bipartite case; see, for example, [GaKa82] or [CoHo82]. For the general case, an algorithm can be found in [HoNS86].

Sometimes the coloring we are looking for does not need to be optimal, but might be slightly worse; then the proof of Vizing's Theorem given above yields an algorithm which with complexity $O(|V||E|)$ finds an edge coloring using just $\Delta(G)+1$ colors.

9.4 Cayley graphs

This section is devoted to a class of graphs which admit a particularly interesting automorphism group. A graph $G = (V, E)$ is called a *Cayley graph* if it has an automorphism group H which operates *regularly* (or *sharply transitively*) on V: for any two vertices v and w, there is exactly one automorphism $h \in H$ which maps v to $v^h = w$.[2] We have two reasons for studying these

[2] We denote the image of a vertex v under some automorphism h by v^h; this is common usage in algebra as well as in finite geometry. Also note that we do not require $H = \mathrm{Aut}\, G$: H is just some subgroup of $\mathrm{Aut}\, G$.

graphs in this section: first, we want to prove something nontrivial about automorphisms of graphs at least once in this book; and second, we will be able to give an interesting application of Vizing's theorem.

Usually, Cayley graphs are defined by an explicit description as follows. Let H be some finite group (written multiplicatively, with unit element 1), and let S be a subset of H having the following properties:

$$1 \notin S \quad \text{and} \quad S = S^{-1} := \{s^{-1} \colon s \in S\}. \tag{9.1}$$

Then we define a graph $G = G(H, S)$ with vertex set $V = H$ and edge set

$$E \;=\; E(H, S) \;=\; \{\{x, y\} \colon xy^{-1} \in S\}.$$

Note that this indeed yields a graph: $xy^{-1} \in S$ is equivalent to $yx^{-1} \in S^{-1}$, and $S = S^{-1}$ holds by (9.1). Now H operates on G by right translation: $h \in H$ maps x to $x^h := xh$. Thus G is a Cayley graph with respect to H. (Note that $h \in H$ maps an edge $\{x, y\}$ to the edge $\{xh, yh\}$, since $xy^{-1} \in S$ is equivalent to $(xh)(yh)^{-1} \in S$.) In fact, every Cayley graph can be written in this form.

Lemma 9.4.1. *A graph $G = (V, E)$ is a Cayley graph with respect to the automorphism group H if and only if G is isomorphic to a graph of the form $G(H, S)$.*

Proof. We have already seen that the condition in the assertion is sufficient. Conversely, let G be a Cayley graph with respect to H. Choose an arbitrary vertex c in G as *base vertex*, and identify each vertex v with the unique element h of H for which $c^h = v$ holds. In particular, c is identified with the unit element 1. Now we define S by

$$S \;=\; \{h \in H \colon \{1, h\} \in E\}.$$

If $\{x, y\}$ is an edge of G, then $\{xy^{-1}, 1\} = \{xy^{-1}, yy^{-1}\}$ is likewise an edge, as H is an automorphism group of G, and as $h \in H$ maps a vertex $z = 1^z$ to $zh = (1^z)^h$. Thus $\{x, y\} \in E$ is equivalent to $xy^{-1} \in S$. If $\{1, h\}$ is an edge, then $\{h^{-1}, 1\}$ is an edge as well, so that $S = S^{-1}$. As G does not contain any loops, $1 \notin S$ is also satisfied, and G is isomorphic to $G(H, S)$. □

Next we determine the connected components of a Cayley graph.

Lemma 9.4.2. *The connected components of $G = G(H, S)$ are precisely the right cosets of the subgroup U of H generated by S.*

Proof. By definition of a Cayley graph, $\{x_i, x_{i+1}\}$ is an edge if and only if $s_i = x_i x_{i+1}^{-1} \in S$ (for $i = 1, \ldots, m-1$). Therefore $(x_1, \ldots, x_m)$ is the sequence of vertices of a walk in G if and only if $x_1 x_m^{-1} = s_1 \ldots s_{m-1}$ is an element of U; that is, iff $x_1 \in U x_m$. □

Let us consider the problem when the chromatic index of a Cayley graph $G(H, S)$ is equal to $\Delta(G)$. Note that a Cayley graph $G(H, S)$ is always regular. Hence Corollary 9.3.3 and Lemma 9.4.2 imply a necessary condition for

$\chi'(G) = \Delta(G)$: the subgroup U of G generated by S has to have even order. Then an edge coloring using $\Delta(G)$ colors is the same as a 1-factorization of G. As we will see, there are reasons to believe that this condition is also sufficient.

Conjecture 9.4.3. A Cayley graph $G(H, S)$ has a 1-factorization if and only if the subgroup U of H generated by S has even order.

Stern and Lenz [StLe80] proved that this conjecture holds for *cyclic* graphs, that is, Cayley graphs $G(H, S)$ for which H is a cyclic group. Later, Stong [Sto85] – who apparently was not aware of the paper of Stern and Lenz – obtained stronger results; we shall present the most important ones. The proof we give is a somewhat simplified variation of the proofs given by these authors; all known proofs rely heavily on Vizing's theorem. The following result gives a general construction for 1-factorizations in certain Cayley graphs, which we will use to prove Conjecture 9.4.3 for three specific classes of groups.[3]

Theorem 9.4.4. *Let $G = G(H, S)$ be a Cayley graph, and suppose that the group H has a normal subgroup N of index 2.*[4] *Then G has a 1-factorization provided that S satisfies one of the following two conditions:*

(1) *$G^* = G(N, S \cap N)$ has a 1-factorization.*
(2) *There is an element $d \in S \setminus N$ such that $s \in S \cap N$ implies $dsd^{-1} \in S \cap N$.*

Proof. If (1) holds, we may assume w.l.o.g. that S is not contained in N, so that we can choose some element $d \in S \setminus N$; and if (2) holds, let d be an element as described there. In either case we have $H \setminus N = dN = Nd$. Consider the cocycle C in G defined by the cut (N, dN):

$$C = \{\{x, y\} \in E\colon x \in N \text{ and } y \in dN\}.$$

Since the subgroup N of H operates regularly (by right translation) both on N and on dN, the orbits of N on C have to be 1-factors, say $F_1, \ldots, F_r$. We may assume $F_1 = \{\{n, dn\} : n \in N\}$, as $n(dn)^{-1}$ is in S. Deleting the remaining F_i $(i = 2, \ldots, r)$ from G yields a regular graph G'. If we delete F_1 as well, we get a disconnected graph G'' which is the disjoint union of the two isomorphic graphs $G^* = G(N, S \cap N)$ and G^*d induced by G on N and on $dN = Nd$, respectively. (Here G^*d denotes the image of G^* under right translation by d.)

Now it suffices to find a 1-factorization of either G' or G''. If condition (1) is satisfied, G^* (and hence also G^*d) has a 1-factorization, which yields a 1-factorization of G''.

[3] For the (elementary) statements and definitions concerning groups which we use in the remainder of this section, we refer the reader to [Hup67] or [Suz82].

[4] This condition holds for large classes of groups as it suffices that all elements of odd order form a normal subgroup of H. For example, this is true if the 2-Sylow subgroups of H are cyclic. A simple ad hoc proof of this statement is given in Lemma X.12.1 of [BeJL99]. For a stronger result in this direction, we refer to [Hup67, Satz IV.2.8].

Finally, suppose that condition (2) is satisfied. We denote the degree of the vertices of the regular graph G^* (and hence also of G^*d) by t; then G' is $(t+1)$-regular. Using Vizing's theorem, we may color the edges of G^* with $t+1$ colors. Note that the mapping $n \mapsto dn$ induces an isomorphism from G^* to G^*d: $\{m,n\}$ is an edge of G^* if and only if $mn^{-1} \in S \cap N$ holds; that is – because of (2) – iff $d(mn^{-1})d^{-1} = (dm)(dn)^{-1} \in S \cap N$. Now this is equivalent to $\{dm, dn\}$ being an edge of G, and hence also of G^*d. We use this isomorphism to define an edge coloring of G^*d with $t+1$ colors: an edge $\{dm, dn\}$ of G^*d is assigned the color of the edge $\{m,n\}$ of G^*. Since both G^* and G^*d are t-regular, there is exactly one color $c(v)$ missing at each of the vertices v of G'. By construction, we have $c(n) = c(dn)$ for all $n \in N$. Thus we may color the edge $\{n, dn\}$ of F_1 using the color $c(n)$ (for each n). In this way, we get an edge coloring of G' with $t+1$ colors; this edge coloring is equivalent to the desired 1-factorization of G'. □

We can now prove Conjecture 9.4.3 for three large classes of groups.

Theorem 9.4.5. *Let H be an abelian group, a 2-group, or a generalized dihedral group. Then the Cayley graph $G = G(H,S)$ has a 1-factorization if and only if the subgroup u of H generated by S has even order.*

Proof. If H is a 2-group (that is, $|H| = 2^a$ for some a), then H has a normal subgroup N of index 2. Using induction on a, we may assume that we know a 1-factorization for $G(N, S \cap N)$ already. Then condition (1) of Theorem 9.4.4 holds, so that G itself has a 1-factorization.

Next suppose that H is abelian. We may assume that G is connected; then $H = U$, and H has even order. Again, H has a normal subgroup of index 2. As G is connected, there exists an element $d \in S \setminus N$. Now condition (2) of Theorem 9.4.4 is satisfied because H is abelian, so that G has a 1-factorization.

Finally, suppose that H is a generalized dihedral group: H has even order $2n$ and is the semi-direct product of an abelian group N of order n with the cyclic group of order 2 (which we write multiplicatively as $\{1,b\}$ here); moreover, the relation $bab = a^{-1}$ holds for all $a \in N$.[5] As every subgroup of H is either abelian or a generalized dihedral group again, we may assume that G is connected: $H = U$. Again, there exists an element $d \in S \setminus N$, say $d = ba$ with $a \in N$. Then $s \in S \cap N$ implies

$$dsd^{-1} \;=\; (ba)s(a^{-1}b) \;=\; bsb \;=\; s^{-1} \in S \cap N.$$

Hence condition (2) of Theorem 9.4.4 holds, and G has a 1-factorization. □

Unfortunately, the results we know so far do not suffice to prove Conjecture 9.4.3 for, say, all nilpotent groups. However, the conjecture is true for this case if S is a minimal generating set for H; see [Sto85]. Theorem 9.4.5 shows that

[5] The classical dihedral groups are the ones where N is cyclic; see, for example, I.9.15 in [Hup67]. The generalized dihedral groups play an important role in reflection geometry; see [Bac89].

the Petersen graph is not a Cayley graph: by Exercise 7.2.8, it does not have a 1-factorization, and the only groups of order 10 are the cyclic group and the dihedral group.

Cayley graphs are of considerable interest; in particular, the conjecture of Lovász [Lov70a] that any connected Cayley graph contains a Hamiltonian cycle has been studied by many authors. This conjecture is still open, although it has been proved for several specific classes of groups; for example, it holds for all abelian groups. A proof of this result – and more on Cayley graphs and automorphism groups of graphs – can be found in the interesting book [Yap86]. It is also of great interest to examine the strongly regular Cayley graphs; for a nice survey of this subject, see [Ma94]. We conclude this section with a basic exercise.

Exercise 9.4.6. Prove that a Cayley graph $G(H, S)$ is strongly regular with parameters (v, k, λ, μ) if and only if both the following conditions are satisfied:

(1) $|H| = v$ and $|S| = k$.

(2) The list of quotients cd^{-1} with $c, d \in S$ and $c \neq d$ contains each element $h \neq 1$ of H either λ or μ times, depending on whether or not h belongs to S.

The set S is then called a *partial difference set*, since H is usually written additively in this context so that (2) turns into a condition on differences.

9.5 The five color theorem

Let us imagine the plane (or a sphere, which makes no difference topologically) dissected by a net of curves in such a way that no point can belong to more than three of the resulting regions; we speak of an (admissible) *map*, *borders*, and *states*. Thus we rule out the possibility of four or more states meeting in a common border point, as is the case for Utah, Arizona, New Mexico, and Colorado. Now let us choose five different colors and paint our map in such a way that no two states with the same color share a common border; then we have an (admissible) *map coloring*. Can this always be done? The answer is *yes*, by the *five color theorem* of Heawood [Hea90] which we will prove in this section.

Let us translate the map coloring problem into the language of graph theory. To this end, we use our map to define a graph $G = (V, E)$ with the states as vertices; two vertices (that is, two states) will be adjacent if and only if they share a common border. It is easy to see that G is a planar graph; to be explicit, we may choose for each state a suitable point in its interior, and realize adjacency by line segments. Obviously, any admissible map coloring corresponds to a coloring of G and, hence, Heawood's five color theorem becomes the assertion that every planar graph G satisfies $\chi(G) \leq 5$.

Example 9.5.1. As a warmup exercise, we shall prove the analogous *six color theorem*. Let G be a planar graph. By Exercise 1.5.14, there is a vertex v of degree at most 5. Using induction, we may assume that $G \setminus v$ can be colored with six colors. This leaves at least one color which does not yet occur at one of the neighbors of v. Hence the given coloring extends to an admissible coloring for G.

The shortest and most elegant proof of the five color theorem is due to Thomassen [Tho94] and actually establishes quite a bit more. We need some further definitions. An (admissible) *list coloring* of a graph $G = (V, E)$ again assigns a *color* to each of its vertices such that adjacent vertices always have different colors; but now each vertex v has its own *list of colors* $C(v)$ from which we have to choose its color $c(v)$. The graph G is called *k-choosable* if it admits a list coloring for every choice of lists $C(v)$ of cardinality $\geq k$ each. The *choosability* or *list coloring number* of G is the smallest k for which G is k-choosable; this parameter is denoted by $\mathrm{ch}(G)$. Clearly, $\mathrm{ch}(G) \geq \chi(G)$; however, in general, the chromatic number and the list coloring number do not agree, as the following simple example shows.

Example 9.5.2. Consider the bipartite graph $K_{3,3}$ which has chromatic number 2 by Example 9.1.1. Let us assign to the three points in each of the two classes forming the partition of V the color lists $\{1, 2\}$, $\{1, 3\}$, and $\{2, 3\}$; then there is no coloring using these lists, as the reader may easily check.

Exercise 9.5.3. Prove that in fact $\mathrm{ch}(K_{3,3}) = 3$. Hint: Use a case distinction according to whether or not two color lists for vertices in the same part of the bipartition have a color in common.

By a result of [ErRT80], there are bipartite graphs with an arbitrarily large choosability; thus k-choosability can indeed be a much stronger requirement than k-colorability. This makes Thomassen's generalization of the five color theorem to list colorings even more remarkable.

Theorem 9.5.4. *Let G be a planar graph. Then* $\mathrm{ch}(G) \leq 5$.

Proof. By assumption, every vertex v of G is associated with a color list $C(v)$ of cardinality ≥ 5. We may assume G to be drawn in the plane. Let

$$C\colon\ v_1 — v_2 — v_3 — \ldots — v_{p-1} — v_p = v_1$$

be the cycle forming the boundary of the unique infinite face – the *outer face* – of the given planar realization of G. If there is an inner face whose boundary is a cycle of length at least 4, we may add new edges to obtain a graph for which all inner faces are bounded by triangles; clearly, it suffices to prove the result in this (possibly harder) case. We make our problem even more difficult by also prescribing the colors $c(v_1)$ and $c(v_2)$, and by requiring merely $|C(v)| \geq 3$ for all $v \in C \setminus \{v_1, v_2\}$. This trick will allow us to prove the result by induction on $n = |V|$.

The starting case $p = n = 3$ (and thus $G = C$) is trivial, as at least one of the colors in $C(v_3)$ has not yet been used for v_1 or v_2. Now let $n \geq 4$. We shall distinguish two cases; see Figure 9.5 for an illustration.

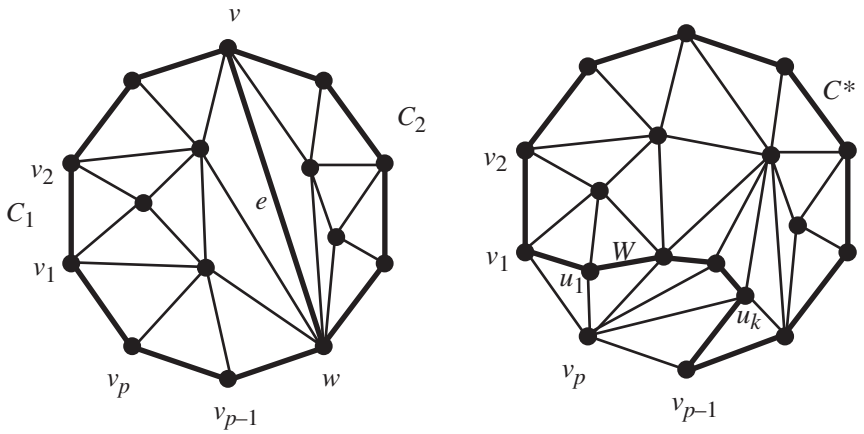


Fig. 9.5. The two cases in the proof of Theorem 9.5.4

First, we assume that C has a chord $e = vw$. Then $C \cup \{e\}$ contains exactly two cycles C_1 and C_2 sharing the edge e; we may assume $v_1v_2 \in C_1$. Now we apply the induction hypothesis to C_1 and obtain a list coloring c_1 of the subgraph with boundary C_1 so that v_1 and v_2 take the prescribed colors. Then we apply the induction hypothesis once again to find a list coloring c_2 of the subgraph with boundary C_2 so that v and w take the same colors as under c_1. Altogether, this gives the desired list coloring of G.

It remains to consider the case where C has no chords. Let us label the neighbors of v_p in their natural order around v_p (say clockwise) as $v_1, u_1, \ldots, u_k, v_{p-1}$; then the u_i lie in the interior of C. As all inner faces of G are bounded by triangles,

$$P\colon\ v_1 — u_1 — u_2 — \ldots — u_k — v_{p-1}$$

has to be a path in G. Since C has no chords, $C^* := P \cup (C \setminus \{v_p\})$ is the boundary cycle of the subgraph $G \setminus v_p$. We choose two colors distinct from $c(v_1)$ in $C(v_p)$ and discard them from all color lists $C(u_i)$ $(i = 1, \ldots, k)$; this leaves shortened color lists of at least three colors each. Now we may apply the induction hypothesis to C^* and the shortened color lists to find a list coloring c of the subgraph $G \setminus v_p$ with boundary C^* for which v_1 and v_2 take the prescribed colors. Finally, we may choose an admissible color for v_p, since at least one of the two colors previously selected in $C(v_p)$ for discarding has to be distinct from $c(v_{p-1})$. □

Corollary 9.5.5 (five color theorem). *Let G be a planar graph. Then $\chi(G) \leq 5$.* □

For more than one hundred years, the most famous open problem in graph theory was the question if four colors suffice for coloring planar graphs (*four color conjecture, 4CC*). This problem was first posed in October 1852 by Francis Guthrie; the first written reference occurs in a letter by Augustus de Morgan to Sir William Rowan Hamilton. In 1878, Arthur Cayley presented the problem to the London Mathematical Society, making it more widely known. A year later, the English lawyer and mathematician Alfred Bray Kempe published a proof for the correctness of the 4CC, see [Kem79]; a decade after that, Heawood [Hea90] discovered a fundamental error in Kempe's arguments. Nevertheless, he managed to modify these arguments to obtain the five color theorem. Subsequently, many important mathematicians worked on the 4CC, among them Heffter, Birkhoff, Ore, and Heesch.

The conjecture was finally proved – more than a century after it had been posed – by Appel and Haaken [ApHa77, ApHK77] with the help of a computer, after a series of theoretical reductions based on the ideas of Kempe and Heesch (*unavoidable configurations*). There was some controversy about the validity of this proof, because of its extraordinary length and complexity as well as the huge amount of computer time needed for establishing the result. In response to this, Appel and Haaken presented a 741 page algorithmic treatment [ApHa89], where they also corrected several minor errors in their original proof. Finally, a much simplified proof (though still using the same basic approach) was given by Robertson, Sanders, Seymour and Thomas [RoSST97]; this also provides an independent verification. For a sketch of this new proof and a discussion of relations between the 4CC and other branches of mathematics, see [Tho98]. A nice account of the 4CC and its solution may be found in the book [Wil2002]. To sum up, the 4CC is now firmly established and may be stated as the following theorem.

Result 9.5.6 (four color theorem). *Every planar graph satisfies $\chi(G) \leq 4$.*

One might hope to prove this result by establishing a strengthening to list colorings, in analogy with Theorem 9.5.4. Unfortunately, this is impossible, as there are planar graphs which are not 4-choosable; the first example for this phenomenon is in [Voi93].

The coloring problem has also been considered (and indeed solved) for maps drawn on other types of surfaces; for this topic, where the fundamental work is due to Gerhard Ringel, we refer the reader to [MoTh01]. We conclude with two more references. For a good survey on list colorings, see [Woo01]. Finally, [Aig84] develops graph theory motivated by the four color conjecture and the attempts to solve it.

10

Circulations

Round and round the circle...
T. S. Eliot

In Chapter 6, we introduced the simplest kind of flow problems, namely the determination of maximal flows in a network; and in Chapter 7, we studied various applications of this theory. The present chapter deals with generalizations of the flows we worked with so far. For example, quite often there are also lower bounds on the capacities of the edges given, or a cost function on the edges. To solve this kind of problem, it makes sense to remove the exceptional role of the vertices s and t by requiring the flow preservation condition (F2) of Chapter 6 for all vertices, including s and t. This leads to the notion of *circulations* on directed graphs. As we will see, there are many interesting applications of this more general concept. To a large part, these cannot be treated using the theory of maximal flows as presented before; nevertheless, the methods of Chapter 6 will serve as fundamental tools for the more general setting.

10.1 Circulations and flows

Let $G = (V, E)$ be a digraph; in general, we tacitly assume that G is connected. A mapping $f : E \to \mathbb{R}$ is called a *circulation* on G if it satisfies the flow conservation condition

(Z1) $\quad \sum_{e^+=v} f(e) \;=\; \sum_{e^-=v} f(e) \quad$ for all $v \in V$.

In addition, let $b : E \to \mathbb{R}$ and $c : E \to \mathbb{R}$ be two further mappings, where $b(e) \leq c(e)$ for all $e \in E$. One calls $b(e)$ and $c(e)$ the *lower capacity* and the *upper capacity* of the edge e, respectively. Then a circulation f is said to be *feasible* or *legal* (with respect to the given capacity constraints b and c) if

(Z2) $\quad b(e) \leq f(e) \leq c(e) \quad$ for all $e \in E$.

Finally, let $\gamma : E \to \mathbb{R}$ be a further mapping called the *cost function*. Then the *cost* of a circulation f (with respect to γ) is defined as

$$\gamma(f) = \sum_{e \in E} \gamma(e) f(e).$$

A feasible circulation f is called *optimal* or a *minimum cost circulation* if $\gamma(f) \leq \gamma(g)$ holds for every feasible circulation g.

It is quite possible that no feasible circulations exist: the capacity restrictions may be contradictory, as the simple example in Figure 10.1 shows, where each edge e is labelled $b(e), c(e)$. We shall obtain a criterion for the existence of feasible circulations in the next section.

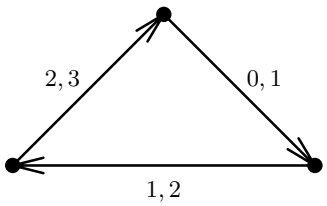


Fig. 10.1. A digraph not admitting any feasible circulation

We now introduce several interesting problems involving circulations. Let us first show that the flows studied in Chapter 6 may be viewed as special types of circulation.

Example 10.1.1 (max-flow problem). Let $N = (G, c, s, t)$ be a flow network with a flow f of value $w(f)$, and let G' be the digraph obtained by adding the edge $r = ts$ to G.[1] We extend the mappings c and f to G' as follows:[2]

$$c(r) = \infty \quad \text{and} \quad f(r) = w(f).$$

Then f is a circulation on G' by (F2) and Lemma 6.1.1. Setting $b(e) = 0$ for all edges e of G', f is even feasible. Conversely, every feasible circulation f' on G' yields a flow of value $f(r)$ on G. The edge r is often called the *return arc* of N. In order to characterize the maximal flows on N in our new terminology, we define a cost function on G' as follows:

$$\gamma(r) = -1 \quad \text{and} \quad \gamma(e) = 0 \quad \text{otherwise.}$$

Then a flow f on N is maximal if and only if the corresponding circulation has minimal cost with respect to γ: maximal flows on $N = (G, c, s, t)$ correspond to optimal circulations for (G', b, c, γ).

---

[1] We may assume, without loss of generality, that ts is not an edge of G; otherwise, we could subdivide ts into two edges.

[2] As usual, ∞ stands for a sufficiently large number, for example $\sum_{e^- = s} c(e)$.

It is not surprising that removing the exceptional role of s and t will allow us to treat circulations in a considerably more elegant manner than flows. For example, it is clear that the set of all circulations on a digraph G' forms a vector space; we will make use of this observation in Section 10.3. Moreover, the fact that circulations are a considerably more general concept enables us to solve a large number of additional problems.

Example 10.1.2 (flows with lower capacity bounds). Let $N = (G, c, s, t)$ be a flow network, and let $b\colon E \to \mathbb{R}$ be an additional mapping satisfying the condition $b(e) \leq c(e)$ for every edge e. We look for a maximal flow f with value $w(f) \geq 0$ on N which satisfies the condition

(F1′) $\quad b(e) \leq f(e) \leq c(e) \quad$ for all $e \in E$,

instead of condition (F1) of Chapter 6. We use the same transformation as in Example 10.1.1: we add the return arc r (with $c(r) = \infty$) and define γ as before. Moreover, we extend the given function b to G' by setting $b(r) = 0$. Again, maximal flows on N correspond to optimal circulations on G'. As before, the problem will be unsolvable should no feasible circulation exist.[3]

Example 10.1.3 (optimal flow problem). Let $N = (G, c, s, t)$ be a given flow network, with an additional cost function $\gamma\colon E \to \mathbb{R}$. The cost of a flow f is defined as for circulations: $\gamma(f) = \sum \gamma(e)f(e)$; in other words, $\gamma(e)$ is the cost resulting from one unit of flow flowing through the edge e. Suppose that the maximal value of a flow through N is w. We require an *optimal flow*, that is, a flow of value w having minimal cost.

In order to formulate this problem in the terminology of circulations, we again introduce the return arc $r = ts$, and put $c(r) = b(r) = w$ and $\gamma(r) = 0$. If there is a lower bound b on the capacities in N (as in Example 10.1.2), we have specified all the necessary data; otherwise, we put $b(e) = 0$ for all edges of G. Now a feasible circulation corresponds to a flow of value w, since $b(r) = c(r) = w$. As $\gamma(r) = 0$, an optimal circulation is the extension of an optimal flow.

Example 10.1.4 (assignment problem). We will show that the assignment problem defined in Example 7.4.12 may be reduced to an optimal flow problem. Let $A = (a_{ij})$ be an $n \times n$ matrix with nonnegative real entries; we require a diagonal D of A for which the sum of the entries in D is minimal. Construct a bipartite digraph G with vertex set $S \dot\cup T$, where $S = \{1, \ldots, n\}$ and $T = \{1', \ldots, n'\}$, and adjoin two additional vertices s and t. The edges of G are all the (s, i), all the (i, j'), and all the (j', t) with $i \in S$ and $j' \in T$; all edges have capacity $c(e) = 1$. Finally, define the cost function

[3] The requirement $b(r) = 0$ guarantees that only flows with a non-negative value correspond to feasible circulations. Note that we need to add the condition $w(f) \geq 0$ to the definition in terms of flows, as trivial examples show. We certainly do not want to consider flows with a negative value, as these would (intuitively) correspond to a flow in the reverse direction, namely from t to s.

by $\gamma(s,i) = \gamma(j',t) = 0$ and $\gamma(i,j') = a_{ij}$. Then the optimal integral flows (with value n) correspond to the solutions of the assignment problem.

As the diagonals of A correspond to the complete matchings of the bipartite graph on $S \dot\cup T$, the assignment problem can also be formulated as a problem concerning weighted matchings; we will study the assignment problem from this point of view in Chapter 14. In §10.10, we will look at some generalizations of the assignment problem.

Before concluding this introductory section with two exercises, we see that even the determination of shortest paths may be formulated in the framework of circulations.

Example 10.1.5 (shortest path problem). Let G be a network where $\gamma(e)$ is the length of the edge e; we require a shortest path from s to t. Interpreting γ as the cost function and assigning capacity $c(e) = 1$ to each edge, a shortest path from s to t is the same as an integral flow of value 1 with minimal cost, so that the problem is a special case of the optimal flow problem. Of course, problems concerning paths are not solved in this way in practice; on the contrary, determining shortest paths is often used as a tool for constructing optimal circulations.

Exercise 10.1.6. Let G be a connected mixed multigraph where each vertex is incident with an even number of edges. Reduce the question whether an Euler tour exists in G (note that all directed edges have to be used according to their direction in such a tour!) to the determination of a feasible circulation in an appropriate digraph; see [FoFu62].

Exercise 10.1.7 (caterer problem). The owner of some restaurant needs fresh napkins every day, say $r_1, \ldots, r_N$ napkins for N consecutive days. He can either buy new napkins (paying some price α for each napkin) or use washed ones; here, the laundry service offers two possibilities: a fast service (the napkins are returned clean after m days at a cost of β per napkin) and a standard service (taking n days at a price of δ for each napkin). All napkins have to be bought before the first day. Formulate the task of supplying the required napkins at the lowest possible cost as an optimal flow problem. The caterer problem has its origin in the practical task of either servicing or buying new engines for airplanes; see [FoFu62], §III.8.

10.2 Feasible circulations

In this section we consider the problem of finding a feasible circulation for a given digraph G with capacity constraints b and c, or to prove the nonexistence of such a circulation. This problem can be solved using the methods of Chapter 6 by considering an appropriate flow network [FoFu62, BeGh62]. To simplify the presentation somewhat, we assume $b(e) \geq 0$ for all edges e; the

general, more technical case can be treated in a similar manner; see Exercise 10.2.3. For many practical applications, the additional condition is satisfied, as indicated by the examples of the previous section.

Let $G = (V, E)$ be a digraph with nonnegative capacity constraints b and c. We add two new vertices s and t and all edges of the form sv or vt to G (where $v \in V$) and denote the resulting larger digraph by H. Now we define a capacity function c' on H as follows:

$$\begin{aligned} c'(e) &= c(e) - b(e) \quad \text{for all } e \in E; \\ c'(sv) &= \sum_{e^+=v} b(e) \quad \text{for all } v \in V; \\ c'(vt) &= \sum_{e^-=v} b(e) \quad \text{for all } v \in V. \end{aligned}$$

This defines a flow network $N = (H, c', s, t)$. Obviously, $c'(e) \geq 0$ for each edge of H. Hence the methods of Chapter 6 can be used to determine a maximal flow on N; let f' be such a flow. Note that the value of f' is at most

$$W = \sum_{v \in V} c'(sv) = \sum_{e \in E} b(e) = \sum_{v \in V} c'(vt).$$

Moreover, $w(f') = W$ holds if and only if f' saturates every edge of H incident with s or t: $f'(sv) = c'(sv)$ and $f'(vt) = c'(vt)$ for all $v \in V$. We now show that there exists a feasible circulation on G if and only if f' achieves this bound.

Theorem 10.2.1. *Let $G = (V, E)$ be a digraph with nonnegative capacity constraints b and c, and let $N = (H, c', s, t)$ be the flow network defined above. Then there exists a feasible circulation on G if and only if the maximal value of a flow on N is $W = \sum_{e \in E} b(e)$.*

Proof. First let f' be a flow of value $w(f') = W$ on N. We define a function f on E by

$$f(e) = f'(e) + b(e) \quad \text{for all } e \in E. \tag{10.1}$$

By definition, f' satisfies the condition $0 \leq f'(e) \leq c'(e) = c(e) - b(e)$; hence f satisfies condition (Z2). It remains to check that f also satisfies (Z1). Thus let v be any vertex of G. As f' is a flow on N, (F2) implies

$$f'(sv) + \sum_{e^+=v} f'(e) = f'(vt) + \sum_{e^-=v} f'(e). \tag{10.2}$$

As $w(f') = W$, all edges of H incident with s or t are saturated, so that

$$f'(sv) = \sum_{e^+=v} b(e) \quad \text{and} \quad f'(vt) = \sum_{e^-=v} b(e). \tag{10.3}$$

Now (10.1) and (10.2) imply (Z1):

$$\sum_{e^+=v} f(e) \;=\; \sum_{e^-=v} f(e). \tag{10.4}$$

Conversely, let f be a feasible circulation on G. Then we can define a mapping f' on the edge set of H using (10.1) and (10.3). As f is feasible, we have $0 \le f'(e) \le c'(e)$ for each edge e of G. For edges of the form sv and vt, we have $c'(sv) = f'(sv)$ and $c'(vt) = f'(vt)$, respectively. Thus all edges incident with s are saturated, and therefore $w(f') = W$. Then f' is indeed a flow, as (10.1), (10.3), and (10.4) imply (10.2). □

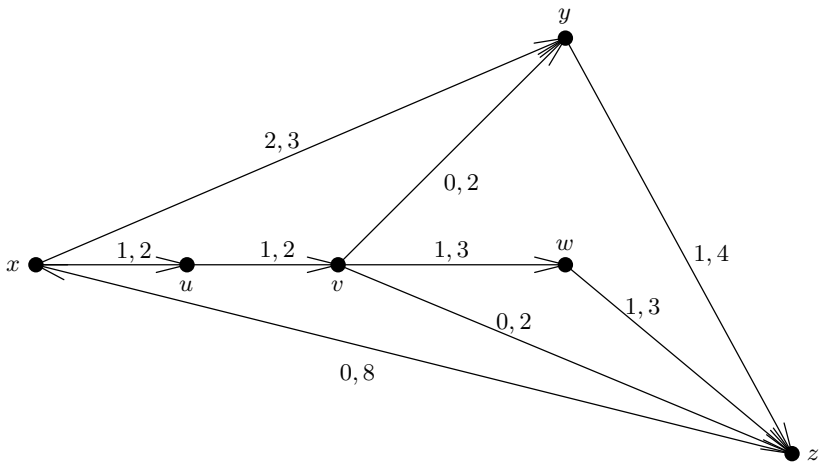


Fig. 10.2. A digraph with capacity constraints

Example 10.2.2. Let G be the digraph given in Figure 10.2 with capacity constraints b and c. We require a feasible circulation on G. By Theorem 10.2.1, we have to determine a maximal flow for the network N shown in Figure 10.3. In general, we would use one of the algorithms of Chapter 6 for such a task; for this simple example, the desired maximal flow can be found by inspection.

As we know that all edges incident with s or t have to be saturated if there exist feasible circulations, we define the value of the flow on these edges accordingly. (The values of f' are printed in bold face in Figure 10.3.) Then (F2) holds for the vertices u, v, and w. As (F2) has to hold for x as well, we put $f'(zx) = 3$; finally, with $f'(yz) = 1$ (F2) holds also for y and z.

From f' we obtain the feasible circulation f on G given in Figure 10.4. Note that f may be interpreted as a feasible flow from x to z having value $w(f) = 3$. This is not yet a maximal feasible flow from x to z: we can increase

the value of the flow to 3 on the path $x - y - z$, and to 2 on the path $x - u - v - w - z$. In this way we obtain a flow of value 5; in view of the capacities of the edges xy and xu this is the maximal possible flow value.

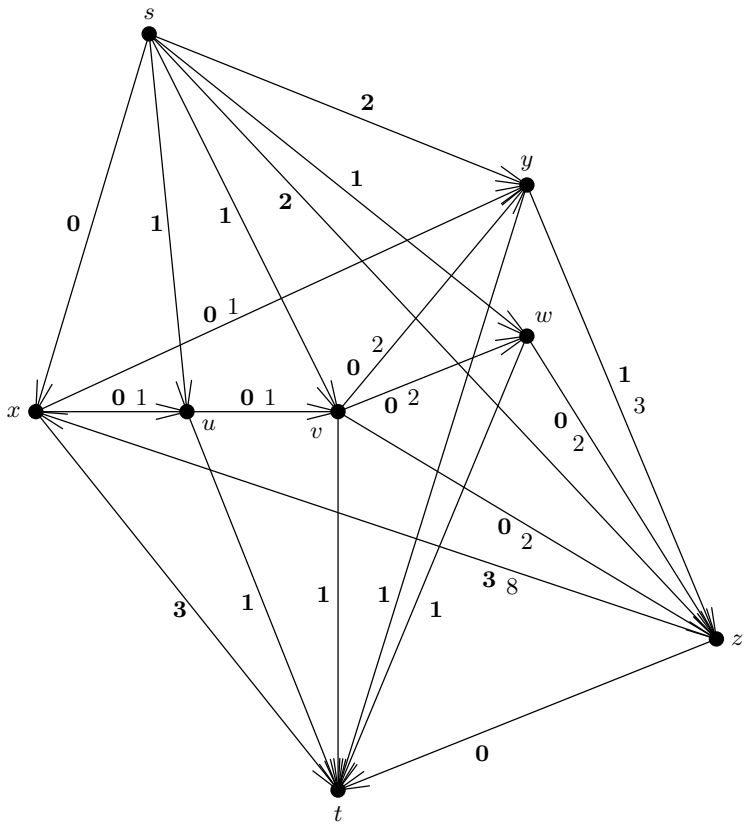


Fig. 10.3. The corresponding flow network

Exercise 10.2.3. Modify the construction of the flow network N for Theorem 10.2.1 so that it applies also to negative lower capacity constraints $b(e)$.

Note that the proof of Theorem 10.2.1 is constructive. Together with Exercise 10.2.3, we obtain the following algorithm for checking if feasible circulations exist, and for determining such a circulation (if possible).

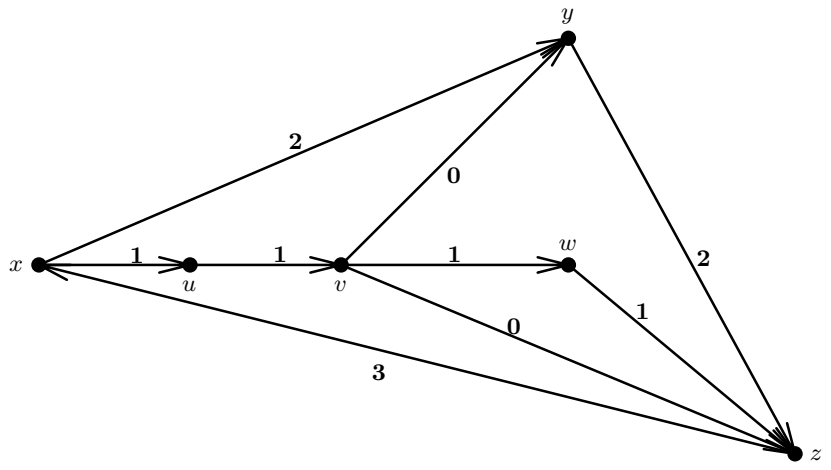


Fig. 10.4. A feasible circulation

Algorithm 10.2.4. Let G be a digraph with capacity constraints b and c.

Procedure LEGCIRC$(G, b, c;$ legal, $f)$

(1) $V' \leftarrow V \cup \{s,t\}$, $E' \leftarrow E \cup \{sv : v \in V\} \cup \{vt : v \in V\}$;
(2) **for** $e \in E$ **do** $c'(e) \leftarrow c(e) - b(e)$ **od**
(3) **for** $v \in V$ **do** $c'(sv) \leftarrow \sum_{e^+=v, b(e)>0} b(e) - \sum_{e^-=v, b(e)<0} b(e)$;

$$c'(vt) \leftarrow \sum_{e^-=v, b(e)>0} b(e) - \sum_{e^+=v, b(e)<0} b(e) \textbf{ od}$$

(4) $H \leftarrow (V', E')$, $N \leftarrow (H, c', s, t)$;
(5) FIFOFLOW $(N; f')$;
(6) **if** $f'(sv) = c'(sv)$ for all $v \in V$ **then** legal $\leftarrow$ true **else** legal $\leftarrow$ false **fi**
(7) **if** legal=true **then**
(8) **for** $e \in E$ **do** $f(e) \leftarrow f'(e) + b(e)$ **od**
(9) **fi**

Corollary 10.2.5. *Algorithm 10.2.4 decides with complexity $O(|V|^3)$ whether there exists a feasible circulation on G; in this case, it also constructs such a circulation.*

Proof. Theorem 10.2.1 and Exercise 10.2.3 show that the algorithm is correct. As the network N has $O(|V|)$ vertices, a maximal flow f' on N can be constructed with complexity $O(|V|^3)$ by Theorem 6.6.15. All the remaining operations in Algorithm 10.2.4 have complexity $O(|E|)$. □

Clearly, we may replace FIFOFLOW in Algorithm 10.2.4 by any other algorithm for finding a maximal flow on N; of course, we then also get the corresponding – in general different – complexity. For example, we may achieve a complexity of $O(|V|^2|E|^{1/2})$; see Theorem 6.6.17.

Exercise 10.2.6. Describe an algorithm which decides whether a given flow network with lower capacity constraint b has a feasible flow and, if possible, constructs a maximal feasible flow; also, discuss the complexity of such an algorithm.

Note that there exists a completely different algorithm for constructing a feasible circulation. We begin with any circulation (for example $f = 0$) and change this circulation successively until we either get a feasible circulation or realize that no such circulation exists. We refer the reader to [FoFu62], §11.3. The algorithm described there has the disadvantage that it is not possible to give a polynomial bound for the complexity, even for integral capacity constraints b and c: the complexity depends on the values b and c take.

Next we give a criterion for the existence of a feasible circulation which may be viewed as a generalization of the max-flow min-cut theorem. We need some notation. Let G be a digraph with nonnegative capacity constraints b and c. A *cut* of G is a partition $V = S \dot\cup T$ of the vertex set of G. The *capacity* of the cut (S, T) is given by

$$c(S,T) \;=\; \sum_{e^-\in S, e^+\in T} c(e) - \sum_{e^+\in S, e^-\in T} b(e).$$

Now consider the flow network $N = (H, c', s, t)$ constructed on G as in Theorem 10.2.1. Then (S', T') with $S' = S \cup \{s\}$ and $T' = T \cup \{t\}$ is a cut of N with capacity

$$c'(S',T') \;=\; \sum_{e^-\in S', e^+\in T'} c'(e),$$

where e runs through all the edges of H. By definition of c', we obtain the following identity, where the sums now run over the edges e of G only:

$$\begin{aligned} c'(S',T') \;&=\; \sum_{v\in T} c'(sv) + \sum_{v\in S} c'(vt) + \sum_{e^-\in S, e^+\in T} c'(e) \\ &=\; \sum_{e^+\in T} b(e) + \sum_{e^-\in S} b(e) + \sum_{e^-\in S, e^+\in T} (c(e) - b(e)) \\ &=\; \sum_{e^-\in S, e^+\in T} c(e) - \sum_{e^+\in S, e^-\in T} b(e) + \sum_{e} b(e) \\ &=\; c(S,T) + W. \end{aligned}$$

Note that every cut (S', T') of N arises from a cut (S, T) of G in this way. By Theorem 10.2.1, there exists a feasible circulation on G if and only if the

maximal value of a flow on N is equal to W; and by Theorem 6.1.6, the maximal value of a flow on N equals the minimal capacity of a cut (S', T'). Thus we get the condition $c'(S', T') \geq W$ for all (S', T'); that is, $c(S, T) \geq 0$ for all cuts (S, T) of G. We have proved the following fundamental result due to Hoffman [Hof60].

Theorem 10.2.7 (circulation theorem). *Let G be a digraph with nonnegative capacity constraints b and c. Then there exists a feasible circulation on G if and only if each cut (S, T) of G has nonnegative capacity, which means*

$$\sum_{e^- \in S, e^+ \in T} c(e) \;\geq \sum_{e^+ \in S, e^- \in T} b(e)$$

for every cut (S, T) of G. □

We may use Theorem 10.2.7 to characterize the maximal value of a feasible flow on a flow network $N = (G, c, s, t)$ with a lower capacity b; cf. Example 10.1.2 and Exercise 10.2.6. Again, we add the return arc $r = ts$ to G and put $b(r) = v$ and $c(r) = \infty$. Then the feasible circulations correspond to feasible flows on N with value $\geq v$. According to Theorem 10.2.7, such a circulation exists if the condition $c(S, T) \geq 0$ holds for every cut (S, T) of G. If $t \in S$ and $s \in T$, the term $c(r) = \infty$ occurs in $c(S, T)$, so that the condition clearly holds for such cuts. If $s \in S$ and $t \in T$, the term $-b(r) = -v$ occurs in $c(S, T)$ which yields the condition

$$\sum_{e^- \in S, e^+ \in T} c(e) - \sum_{e^+ \in S, e^- \in T} b(e) \;\geq\; v.$$

In the remaining cases – that is, for $s, t \in S$ or $s, t \in T$ – we get conditions which do not involve the value v of the flow, since the return arc r does not occur in the sum for $c(S, T)$; these conditions are needed to ensure the existence of some feasible flow. Thus we get the maximal value for a flow – assuming that there actually exist feasible flows on N – if v is the minimal capacity of an *ordinary* cut of N: a cut with $s \in S$ and $t \in T$. Here, of course, we have to define the capacity $c(S, T)$ as before:

$$c(S, T) \;= \sum_{e^- \in S, e^+ \in T} c(e) - \sum_{e^+ \in S, e^- \in T} b(e). \tag{10.5}$$

We have proved the following result.

Theorem 10.2.8. *Let $N = (G, b, c, s, t)$ be a flow network with a nonnegative lower capacity b. The following is a necessary and sufficient condition for the existence of feasible flows on N:*

$$c(X, Y) \geq 0 \quad \textit{for all partitions } V = X \,\dot\cup\, Y \textit{ with } t \notin X \textit{ or } s \notin Y. \tag{10.6}$$

If (10.6) holds, the maximal value of a feasible flow equals the minimum of the capacities $c(S, T)$ (defined as in (10.5)) over all cuts (S, T). □

In the special case of ordinary flows (that is, for $b \equiv 0$), the existence of feasible flows is trivial, and Theorem 10.2.8 reduces to the max-flow min-cut theorem.

Exercise 10.2.9. Let G be a mixed multigraph. Find necessary and sufficient conditions for the existence of an Euler tour in G; cf. Exercise 10.1.6.

Exercise 10.2.10. Let $N = (G, b, c, s, t)$ be a flow network with a nonnegative lower capacity b. Describe a technique for determining a *minimal* feasible flow on N (that is, a feasible flow of minimum value), and discuss its complexity. Moreover, give a description for the value of such a minimal feasible flow analogous to Theorem 10.2.8. (Note that this problem is irrelevant for the ordinary flows treated in Chapter 6: trivially, the zero flow is a minimal feasible flow in that situation.)

Exercise 10.2.11. Let G be a connected digraph with capacity constraints b and c, where $b(e)$ is always positive and $c(e) = \infty$ for all edges e. Show that G has a feasible circulation if and only if it is strongly connected. Moreover, give a criterion for the existence of a feasible flow if we also specify a source s and a sink t. Hint: Use Exercise 8.5.3.

Feasible circulations on undirected and mixed graphs are studied in [Sey79] and [ArPa86], respectively.

10.3 Elementary circulations

In this section, we consider the problem of decomposing a given circulation into circulations which are as simple as possible. The results obtained will be applied in Sections 10.4 to 10.9 when we present three algorithms for determining optimal circulations.

We begin by translating the notion of a circulation into the terminology of linear algebra. Let G be a (not necessarily connected) digraph with incidence matrix M, say with vertex set $V = \{1, \ldots, n\}$ and edge set $E = \{e_1, \ldots, e_m\}$. Every mapping $f\colon E \to \mathbb{R}$ induces a vector $\mathbf{f}$ in $\mathbb{R}^m$, namely

$$\mathbf{f} \;=\; (f(e_1), f(e_2), \ldots, f(e_m))^T.$$

Note that f satisfies condition (Z1) if and only if $M\mathbf{f} = \mathbf{0}$; this follows by recalling that the i-th row of M has entry $+1$ or -1 in those columns j for which the edge e_j has end vertex or start vertex i, respectively. Thus we have the following simple but important result.

Lemma 10.3.1. *Let G be a digraph with incidence matrix M. Then $f\colon E \to \mathbb{R}$ is a circulation if and only if $M\mathbf{f} = \mathbf{0}$.* □

In other words, the circulations are precisely the mappings which are associated with the elements in the kernel of the linear mapping from $\mathbb{R}^m$ to $\mathbb{R}^n$ corresponding to the matrix M. These mappings form a vector space of dimension $m - \operatorname{rank} M$. By Theorem 4.2.4, $\operatorname{rank} M = n - p$, where p is the number of connected components of G. This gives the following result.

Theorem 10.3.2. *Let G be a digraph with incidence matrix M. Then the circulations on G form a vector space of dimension $\nu(G) = m - n + p$; here m, n, and p denote, respectively, the number of edges, vertices, and connected components of G.* □

Corollary 10.3.3. *Let f be a circulation on the digraph G. If G is a tree, then necessarily $f = 0$.*

Proof. For trees, $p = 1$ and $m = n - 1$, hence $\nu(G) = 0$. □

Exercise 10.3.4. Let (S, T) be a cut of G and f a circulation on G. Show $f(S, T) = f(T, S)$, where

$$f(S,T) \;=\; \sum_{e^- \in S, e^+ \in T} f(e).$$

Also prove – without using algebraic tools – that the support of a circulation (compare Exercise 6.1.13) cannot contain any bridges.

Next we look for canonical bases for the vector space of circulations on G. We need a definition. A circulation $f \neq 0$ is called *elementary* if its support is minimal with respect to inclusion: there is no circulation $g \neq 0$ for which the support of g is contained in, but not equal to, the support of f. The following result shows that elementary circulations correspond to cycles.

Lemma 10.3.5. *Let G be a digraph and f a circulation on G. Then f is elementary if and only if the support of f is a cycle of G. For every cycle C of G, there exists an elementary circulation with support C.*

Proof. Let $C = (e_1, \ldots, e_k)$ be any cycle. We first construct an elementary circulation f_C with support C: we set $f_C(e) = 0$ for all edges $e \notin C$ and $f_C(e_i) = +1$ or -1 (for $i = 1, \ldots, k$), according as e_i is a forward or a backward edge of C. It is immediate that f_C is a circulation on G. (This is merely the first part of the proof of Theorem 4.2.3 put into different language.)

Now let g be a circulation whose support is strictly contained in the support of f. We may assume $g(e_k) = 0$. As C is a cycle, the edges $e_1, \ldots, e_{k-1}$ form a path; thus Corollary 10.3.3 implies $g = 0$. Hence f_C is elementary. Conversely, let f be any elementary circulation on G. As $f \neq 0$, the support of f has to contain a closed trail and hence a cycle C. Since there exists a circulation having support C and as f is elementary by hypothesis, the support of f has to be C itself. □

The next theorem shows that $\nu(G)$ equals the maximal number of linearly independent cycles of G; this explains why this parameter is usually called the *cyclomatic number* of G.

Theorem 10.3.6. *Let G be a digraph with n vertices, m edges, and p connected components. Then there exists a basis of the vector space V of circulations on G which consists of $\nu(G) = m - n + p$ elementary circulations.*

Proof. It suffices to prove the assertion for each connected component of G; thus we may assume $p = 1$, so that G is connected. Let T be a spanning tree for $|G|$. For each edge e of $|G| \setminus T$, let $C_T(e)$ be the unique cycle of $|G|$ containing e and edges of T only, as in Section 4.3. By Lemma 10.3.5, there exists an elementary circulation f_e on G having support $C_T(e)$. It remains to show that the f_e form a basis of V. In view of Corollary 10.3.2, it suffices to check that the f_e are linearly independent, since there are exactly $m - n + 1$ edges e in $|G| \setminus T$. But this is clear: the support of f_e contains just one edge outside T, namely e. □

Exercise 10.3.7. Write an arbitrary circulation on G explicitly as a linear combination of the elementary circulations f_e constructed in the proof of Theorem 10.3.6.

In view of Lemma 10.3.5 and Theorem 10.3.6, the vector space V of all circulations on G is also called the *cycle space* of G.[4] When weights are assigned to the edges of G, a basis of V having smallest weight can be found with complexity $O(|V||E|^3)$; in the unweighted case, at most $3(n-1)(n-2)/2$ edges are needed; see [Hor87]. However, determining a basis of V having smallest weight and consisting of elementary cycles (that is, a basis as given in the proof of Theorem 10.3.6) is an NP-hard problem; see [DePK82].

Exercise 10.3.8. Let $G = (V, E)$ be a digraph having n vertices, m edges, and p connected components. Let M be the incidence matrix of G, and let $q\colon V \to \mathbb{R}$ be a mapping which we call a *potential*. We define $\delta q\colon E \to \mathbb{R}$ by

$$\delta q(xy) \;=\; q(y) - q(x) \quad \text{for } xy \in E.$$

Any mapping of the form δq is called a *potential difference*; this terminology comes from electricity networks. Show the following results, which are analogous to the preceding results about circulations:

(a) The potential differences form a vector space P corresponding to the row space of M. Determine $\dim P$.
(b) Given any cocycle of G, there exists a potential difference having this cocycle as support.

[4] The cycle space of a graph is a special case of a more general construction assigning certain modules to any *geometry*; see [GhJu90] and the references given there.

(c) If G is connected, use a spanning tree T for constructing a basis of P. Hint: Compare Lemma 4.3.2.

The vector space P is called the *cocycle space* or *bond space* of G, since cocycles are sometimes also called *bonds*.

By Theorem 10.3.6, every circulation on a digraph G can be written as a sum of elementary circulations. In the remainder of this section, we will establish a stronger result: nonnegative circulations can be written as linear combinations of nonnegative elementary circulations with positive coefficients. To this end, we first prove a lemma about colorings due to Minty [Min66], which turns out to be very useful.

Theorem 10.3.9 (painting lemma). *Let G be a digraph whose edges are colored arbitrarily with the colors black, red, and green; edges without color are allowed as well. Moreover, let e_0 be a black edge of G. Then we have one (and only one) of the following alternatives:*

(1) *There exists a cycle K through e_0 which contains no uncolored edges; moreover, all black edges of K have the same orientation as e_0, and all green edges have opposite orientation.*

(2) *There exists a cocycle C through e_0 which does not contain any red edges; moreover, all black edges of C have the same orientation as e_0, and all green edges have opposite orientation.*

Proof. Let $e_0 = ts$. We mark the vertices of G according to the following two rules:

(a) s is marked.

(b) Suppose v is already marked. A vertex u not yet marked is marked if and only if there exists a black or red edge vu, or a red or green edge uv.

This process terminates when no further vertices can be marked according to rule (b). This can happen in two possible ways:

Case 1: t has been marked. Then rule (b) implies the existence of a path from s to t which contains no uncolored edges, and for which each black edge is a forward edge and each green edge a backward edge. Adding the edge $e_0 = ts$ yields a cycle as in alternative (1).

Case 2: t has not been marked. Let S be the set of all vertices which have been marked; then $s \in S$ and $t \in V \setminus S$. Let C be the cocycle of G corresponding to the cut $(S, V \setminus S)$. By rule (b), C cannot contain any red edge, or any black edge with start vertex in S, or any green edge with end vertex in S; for otherwise it would be possible to label a vertex of $V \setminus S$. Thus C is a cocycle as in alternative (2).

It remains to show that (1) and (2) cannot hold simultaneously. Suppose C is a cocycle as in (2), and K is a cycle as in (1). As C and K both contain e_0, K has to contain a further edge e_1 of C, because s and t are in different parts

of the cut (S,T) defining C. We may assume – in case that C and K have more than two edges in common – that e_1 is the next edge in K which also belongs to C, where we traverse K starting with e_0 (in the direction given by the orientation of e_0). As $e_1 \in C \cap K$, this edge cannot be red or uncolored. Suppose first that e_1 is black. Note that $e_1 \in C \cap K$ implies that e_1 has the same orientation as e_0 both in K and in C. However, this is easily seen to be a contradiction: our choice of e_1 as the next edge in K which also belongs to C implies that the orientation of e_1 in C is opposite to that of e_0. For instance, if e_0 is directed from S to T, then e_1 has to be directed from T to S. Finally, if e_1 is green, we again obtain a contradiction, using a similar argument. □

Coloring all edges of G black yields an interesting corollary. We need a notation. Let C be the cocycle corresponding to the cut (S,T). C is called a *directed cocycle* or a *cocircuit* if all edges of C have the same orientation (from S to T, say).

Corollary 10.3.10. *Each edge of a digraph is either contained in a directed cycle or in a directed cocycle.* □

We can now prove the promised result about nonnegative circulations:

Theorem 10.3.11. *Let G be a digraph and $f \neq 0$ a circulation on G. Then f is nonnegative (that is, $f(e) \geq 0$ for all edges e) if and only if f can be written in the form $f = \lambda_1 f_1 + \ldots + \lambda_k f_k$, where the f_i are nonnegative elementary circulations and the λ_i positive numbers.*

Proof. Obviously, the condition stated in the assertion is sufficient. Conversely, let $f \neq 0$ be a nonnegative circulation on G, and let G' be the sub-digraph of G defined by the support of f. As f satisfies condition (Z1), G' cannot contain a directed cocycle. Hence Corollary 10.3.10 guarantees the existence of a directed cycle C_1 in G'. Let f_1 be the elementary circulation corresponding to C_1; that is, $f_1(e) = 1$ for all edges e in C_1, and $f_1(e) = 0$ otherwise. Put $\lambda_1 = \min\{f(e) \colon e \in C_1\} > 0$ and $g = f - \lambda_1 f_1$. Then g is again a nonnegative circulation on G. If $g = 0$, we are done; otherwise the support of g contains at least one edge less than the support of f does, so that the assertion follows by induction. □

Corollary 10.3.12. *Let $N = (G, c, s, t)$ be a flow network. Then any flow can be written as a sum of elementary flows and nonnegative elementary circulations.*

Proof. Let G' be the graph we obtain by adding the return arc $r = ts$ to G; see Example 10.1.1. We know that every flow f can be extended to a circulation on G' by defining $f(r) := w(f)$. As flows are nonnegative by definition, this circulation is likewise nonnegative. Now the assertion follows from Theorem 10.3.11; note that all those elementary circulations f_i on G' whose support contains r yield elementary flows on N. □

As shown in Exercise 6.1.13, one cannot avoid using elementary circulations – and not just elementary flows – in Corollary 10.3.12.

We now translate the proof of Minty's painting lemma into an algorithm which constructs – given any coloring of a digraph with the colors black, red, and green – a cycle K or a cocycle C as described in Theorem 10.3.9. The Boolean variable *cycle* will have value *true* if and only if the algorithm actually constructs a cycle.

Algorithm 10.3.13. Let $G = (V, E)$ be a digraph; F a partial coloring of the edges of G with the colors black, red, and green; and $e_0 \in E$ a black edge.

Procedure MINTY$(G, F, e_0;$ cycle$, K, C)$

(1) **for** $v \in V$ **do** $u(v) \leftarrow$ false **od**
(2) $s \leftarrow e_0^+$; $t \leftarrow e_0^-$; $K \leftarrow \emptyset$; $C \leftarrow \emptyset$; $A \leftarrow \emptyset$;
(3) label s with $(-,-)$ and set $A \leftarrow A \cup \{s\}$;
(4) **repeat**
(5) choose a labelled vertex v with $u(v) =$ false;
(6) **for** $e \in \{e \in E : e^- = v\}$ **do**
(7) **if** $w = e^+$ is not labelled **and** $F(e) =$ red **or** $F(e) =$ black
(8) **then** label w with $(v, +)$; $A \leftarrow A \cup \{w\}$ **fi**
(9) **od**
(10) **for** $e \in \{e \in E : e^+ = v\}$ **do**
(11) **if** $w = e^-$ is not labelled **and** $F(e) =$ red **or** $F(e) =$ green
(12) **then** label w with $(v, -)$; $A \leftarrow A \cup \{w\}$ **fi**
(13) **od;**
(14) $u(v) \leftarrow$ true;
(15) **until** t is labelled **or** $u(v) =$ true for all $v \in A$;
(16) **if** t is labelled
(17) **then** cycle $\leftarrow$ true, $K \leftarrow K \cup \{e_0\}$, $w \leftarrow t$;
(18) **while** $w \neq s$ **do**
(19) find the first component v of the label of w;
(20) **if** the second component of the label of w is $+$
(21) **then** $e \leftarrow vw$ **else** $e \leftarrow wv$ **fi**
(22) $K \leftarrow K \cup \{e\}$; $w \leftarrow v$
(23) **od**
(24) **else** cycle $\leftarrow$ false;
(25) $C \leftarrow \{e \in E\colon\ e^- \in A, e^+ \in V \setminus A$ or $e^+ \in A, e^- \in V \setminus A\}$
(26) **fi**

The following result is immediate from Theorem 10.3.9.

Theorem 10.3.14. *Algorithm 10.3.13 constructs with complexity $O(|E|)$ a cycle or a cocycle of G, as described in Theorem 10.3.9.* □

Example 10.3.15. Let us show that the labelling algorithm of Ford and Fulkerson (Algorithm 6.1.7) may be viewed as a special case of Algorithm 10.3.13. We choose for e_0 the return arc $r = ts$ introduced in Example 10.1.1, and color e_0 black. The remaining edges e are colored as follows: black if e is void, green if e is saturated, and red otherwise. It should be clear that case (1) of the painting lemma then yields an augmenting path from s to t, whereas in case (2) no such path exists. Then the cocycle constructed by the algorithm corresponds to a minimal cut of N (with capacity equal to the value of the maximal flow f).

As in the special situation of Chapter 6, it is advisable to make step (5) of Algorithm 10.3.13 deterministic by always selecting the vertex v with $u(v) =$ false which was labelled first; to achieve this, the labelled vertices are put into a queue. This guarantees that we obtain a shortest path from s to t in case (1) of the painting lemma – that is, a cycle C of shortest length through e_0.

Exercise 10.3.16. Let us sketch a method – which is attributed to Herz [Her67] in [GoMi84] – for determining a first legal circulation in the case of integral capacities. Start with any circulation f. For every edge e violating the feasibility condition, put

$$d(e) := \begin{cases} b(e) - f(e) & \text{if } f(e) < b(e) \\ f(e) - c(e) & \text{if } f(e) > c(e) \end{cases},$$

and $d(e) = 0$ for all other edges; thus the sum D of all $d(e)$ measures how severely f violates the feasibility condition. In particular, this sum is 0 if and only if f is feasible. As long as $D > 0$, we choose any edge e_0 with $d(e) > 0$ and use the painting lemma to either find a better circulation, or to conclude that there are no feasible circulations. Do so in case $f(e) < b(e)$; the case $f(e) > c(e)$ is similar.

Hint: Leave all edges satisfying $b(e) = f(e) = c(e)$ uncolored, and color the remaining edges as follows: black, if $f(e) \leq b(e)$; green, if $c(e) \leq f(e)$; red, if $b(e) < f(e) < c(e)$. The first case in the painting lemma leads to a better circulation, while in the second case no feasible circulations exist; to see this, use Exercise 10.3.4 together with the necessary condition in the circulation theorem 10.2.7.

Exercise 10.3.17. Use Exercise 10.3.16 to give a constructive proof for the circulation theorem 10.2.7 in the case of integral capacities.

10.4 The algorithm of Klein

In the next few sections, we will present three algorithms for constructing optimal circulations. We begin with a particularly simple algorithm which is due

to Klein [Kle67]. In the course of this algorithm, we have to examine an appropriate auxiliary digraph check whether it contains cycles of negative length (and construct such a cycle); we will use Algorithm 3.10.1 (NEGACYCLE) for this purpose.

Algorithm 10.4.1. Let G be a digraph with capacity constraints b and c and a cost function γ. The algorithm checks whether there exists a feasible circulation; if this is the case, an optimal circulation is constructed.

Procedure KLEIN$(G, b, c, \gamma; \text{legal}, f)$

(1) LEGCIRC$(G, b, c; \text{legal}, f)$;
(2) **if** legal = true **then repeat**
(3) $E' \leftarrow \emptyset$;
(4) **for** $e = uv \in E$ **do**
(5) **if** $f(e) < c(e)$
(6) **then** $E' \leftarrow E' \cup \{e\}$; $tp(e) \leftarrow 1$; $c'(e) \leftarrow c(e) - f(e)$;
$w(e) \leftarrow \gamma(e)$ **fi**
(7) **if** $b(e) < f(e)$
(8) **then** $e' \leftarrow vu$; $E' \leftarrow E' \cup \{e'\}$; $tp(e') \leftarrow 2$;
$c'(e') \leftarrow f(e) - b(e)$; $w(e') \leftarrow -\gamma(e)$
(9) **fi**
(10) **od**
(11) $H \leftarrow (V, E')$;
(12) NEGACYCLE$(H, w; d, p, \text{neg}, C)$;
(13) **if** neg = true
(14) **then** $\delta \leftarrow \min\{c'(e) : e \in C\}$;
(15) **for** $e \in C$ **do**
(16) **if** $tp(e) = 1$ **then** $f(e) \leftarrow f(e) + \delta$ **else** $f(e) \leftarrow f(e) - \delta$ **fi**
(17) **od**
(18) **fi**
(19) **until** neg = false
(20) **fi**

It is usual to refer to a change of f along a cycle C as in steps (14) to (17) above simply by saying that the *cycle C is cancelled*.

We now have to check whether Algorithm 10.4.1 is correct, and determine its complexity. First, we show that the algorithm terminates if and only if the circulation f constructed so far is optimal. Step (19) has the effect that the algorithm terminates only if there is no cycle of negative length in the auxiliary digraph H corresponding to f. Thus we have to prove the following result.

Lemma 10.4.2. *Let G be a digraph with capacity constraints b and c and a cost function γ. Moreover, let f be a feasible circulation on G. Then f is optimal if and only if the auxiliary network (H, w) constructed in steps (3)*

to (11) of Algorithm 10.4.1 does not contain any directed cycle of negative length.

Proof. It is clear that the condition given in the assertion is necessary: if (H, w) contains a directed cycle C of negative length, then it is possible to change the present feasible circulation f according to steps (14) to (17) so that we get a new feasible circulation with smaller cost. Note that the cost is changed by the following amount:

$$\delta\Big(\sum_{e\in C, tp(e)=1}\gamma(e) - \sum_{e\in C, tp(e)=2}\gamma(e)\Big) \;=\; \delta\sum_{e\in C} w(e) \;<\; 0.$$

Conversely, assume that the condition of the theorem is satisfied; we have to show $\gamma(g) \geq \gamma(f)$ for every feasible circulation g. To do so, we consider the circulation $g - f$. This circulation induces a circulation h on H as follows:

- if $c(e) \geq f(e) > g(e) \geq b(e)$, we define $h(e') = f(e) - g(e)$, where $e = uv$ and $e' = vu$;
- if $c(e) \geq g(e) > f(e) \geq b(e)$, we set $h(e) = g(e) - f(e)$;
- for all other edges of H, we put $h(e) = 0$.

As h is a nonnegative circulation on H, we may apply Theorem 10.3.11: there exist nonnegative elementary circulations $h_1, \ldots, h_k$ and positive numbers $\lambda_1, \ldots, \lambda_k$ with $h = \lambda_1 h_1 + \ldots + \lambda_k h_k$. Let $w(h)$ denote the cost of the circulation h on H with respect to the cost function w. Then

$$\gamma(g) - \gamma(f) \;=\; w(h) \;=\; \lambda_1 w(h_1) + \ldots + \lambda_k w(h_k) \;\geq\; 0,$$

since (H, w) does not contain any directed cycles of negative length. Hence $\gamma(g) \geq \gamma(f)$, as claimed. □

In step (1) of Algorithm 10.4.1, we apply the procedure LEGCIRC. By Corollary 10.2.5, either a feasible circulation is constructed, or the algorithm terminates (because no such circulation exists so that the Boolean variable *legal* takes the value *false*). We have already indicated that each iteration of the **repeat**-loop changes the present feasible circulation so that the new feasible circulation has smaller cost; we leave the details to the reader.

It remains to address the question under which conditions we can guarantee that the algorithm actually terminates (with an optimal circulation, by Lemma 10.4.2). Let us assume that b and c are integral. Then the original feasible circulation constructed by LEGCIRC is integral as well, because MAXFLOW constructs an integral maximal flow whenever the given capacity constraints are integral; compare Theorem 6.1.5. Therefore the capacity function c' on E' defined in steps (4) to (10) is integral, too, so that δ is integral in step (14). It follows by induction that each feasible circulation constructed during the course of the algorithm is integral. We also know that changing the current circulation f in step (16) by cancelling a cycle C of negative length

decreases the cost by $|\delta w(C)|$; see the proof of Lemma 10.4.2. Thus, if we assume γ to be integral as well, the cost is decreased with each iteration of the **repeat**-loop by a positive integer. Note that

$$m = \sum_{\substack{e \\ \gamma(e) > 0}} \gamma(e)b(e) + \sum_{\substack{e \\ \gamma(e) < 0}} \gamma(e)c(e)$$

is a lower bound on $\gamma(f)$ for every feasible circulation f on G; hence the algorithm has to terminate. This also applies in the case of rational values of b, c, and γ, as these may be multiplied by their common denominator. We have proved the following result.

Theorem 10.4.3. *Let G be a digraph, and assume that the capacity constraints b and c as well as the cost function γ take rational values only. Then Algorithm 10.4.1 determines an optimal circulation f on G. If b, c, and γ are actually integral, f is integral as well.* □

The call of LEGCIRC in Algorithm 10.4.1 has complexity $O(|V|^3)$ by Corollary 10.2.5. Moreover, each iteration of the **repeat**-loop likewise has complexity $O(|V|^3)$, because NEGACYCLE has this complexity by Theorem 3.10.2. Unfortunately, the entire algorithm is, in general, not polynomial: the number of iterations depends on the values of the functions b, c, and γ. However, the algorithm becomes polynomial provided that the cycles of negative length are chosen in an appropriate way; this result is due to Goldberg and Tarjan [GoTa89] and will be proved in Section 10.9.

Exercise 10.4.4. Give an upper bound for the number of iterations of the **repeat**-loop in Algorithm 10.4.1 if the functions b, c, and γ are integral.

Foe a long time, the most popular algorithm for determining an optimal circulation was not the algorithm of Klein presented here, but the so-called *out-of-kilter algorithm*; see [Ful61] and [Min60]. However, that algorithm is considerably more involved; it is based on Minty's painting lemma. We refer the interested reader to [FoFu62], [Law76], or [GoMi84] for a presentation of the out-of-kilter algorithm. It is also not polynomial in our terminology: its complexity likewise depends on the capacity constraints.[5]

[5] We note that the out-of-kilter algorithm *is* polynomial if we include the capacities in the calculation of the size of the input data: for a natural number z, we may take $\log_2 z$ as a measure for the size of z. Algorithms which are polynomial in our sense (that is, their complexity is independent of the capacity constraints and the cost function) are often called *strongly polynomial* in the literature. For this property, all numbers occurring during the algorithm have to be polynomial in the total size of the input data. This is trivially true if the algorithm involves only additions, subtractions, comparisons, and multiplications or divisions by a constant factor. We will sometimes call algorithms which are polynomial in $|E|$, $|V|$, and the logarithm of the size of the input data *weakly polynomial*.

Using an appropriate scaling, a complexity of $O(|E|^2 p)$ can be obtained, where $p = \log_2 C$ and C is the maximum of the capacities $c(e)$; see [EdKa72]. A particularly simple weakly polynomial algorithm is in [BaTa89]; it is based on Theorem 7.5.2 and an idea of Weintraub [Wei74], and uses several cycles of negative length simultaneously during each iteration. The first algorithm which is polynomial in our sense was given by Tardos [Tar85]; the complexity of her algorithm is $O(|E|^2 T \log |E|)$, where T denotes the complexity of the MAXFLOW-routine used. This result was improved and varied several times; we refer the interested reader to [Fuj86, GaTa88, GaTa89, GoTa89, GoTa90, AhGOT92, Orl93]. Altogether, it is possible to reach a complexity of $O(|V|^4 \log |V|)$. The algorithm of Goldberg and Tarjan [GoTa90] will be presented in Section 10.7; it has a complexity of $O(|V|^3 |E| \log |V|)$.

10.5 The algorithm of Busacker and Gowen

In this section, we consider the special case where the lower capacity constraint on G is always 0. In this case, the zero circulation is feasible; if we assume that G does not contain any directed cycles of negative length with respect to γ, it is even optimal. If we consider the edge ts as the return arc, the zero flow is a flow of minimal cost on the flow network $(G \setminus ts, c, s, t)$. We shall now solve the optimal flow problem of Example 10.1.3 by constructing flows of minimal cost with increasing values, beginning with the zero flow. This can be done by using a path of minimal cost for augmenting the flow, as suggested by Busacker and Gowen [BuGo61].

The algorithm of Busacker and Gowen is basically the same as the algorithm of Ford and Fulkerson of Section 6.1, only that each change of the flow is made using an augmenting path of minimal cost. To determine such a path, the auxiliary network introduced at the beginning of Section 6.3 is used. Let $N = (G, c, s, t)$ be a flow network with cost function γ which does not contain any cycles of negative length. Also, let f be an *optimal flow* of value $w(f) = w$ on N; that is, f has minimal cost $\gamma(f)$ among all flows of value w. Now consider the auxiliary network $N' = (G', c', s, t)$ with respect to f and define a cost function γ' on N' as follows:

- for each edge $e = uv$ of G with $f(e) < c(e)$, the edge $e' = uv$ of G' is assigned cost $\gamma'(e') = \gamma(e)$ and capacity $c'(e') = c(e) - f(e)$;
- for each edge $e = uv$ of G with $f(e) > 0$, the edge $e'' = vu$ of G' is assigned cost $\gamma'(e'') = -\gamma(e)$ and capacity $c'(e'') = f(e)$.

Moreover, we add the return arc $r = ts$ to N; put $b(r) = c(r) = w$, $\gamma(r) = 0$, and $f(r) = w$; and $b(e) = 0$ for all other edges e. Then f becomes an optimal circulation, and N' is the auxiliary network corresponding to this circulation. As f is optimal, N' does not contain any directed cycles of negative length, by Lemma 10.4.2. Therefore it is possible – assuming that f is not yet a maximal flow – to find an augmenting path P of minimal cost among all augmenting

paths from s to t in N'; for example, we may use the algorithm of Floyd and Warshall for this task. Denote the capacity of P by δ; then we can use P to construct a flow f' on N of value $w(f') = w + \delta$ (as in Algorithm 6.1.7). We will show that f' is an optimal flow for this value.

To this end, we consider both f' and f as circulations. Then $f' - f$ is a circulation whose support is a cycle C which contains the return arc r and has minimal cost with respect to γ', where $\gamma'(r) = 0$. Thus $f' - f = \delta f_C$, where f_C is the elementary circulation corresponding to the cycle C (as in Theorem 10.3.5). Now suppose that g is any flow of value $w + \delta$ on N, and consider g likewise as a circulation. By analogy to the proof of Lemma 10.4.2, we can show that $g - f$ induces a nonnegative circulation h on N' – more precisely, on G' with the return arc r added. By Theorem 10.3.11, we may write h as a linear combination $h = \lambda_1 h_1 + \ldots + \lambda_k h_k$ of nonnegative elementary circulations on N' with positive coefficients λ_i. We may assume that the h_i are numbered so that the supports of $h_1, \ldots, h_p$ contain the return arc r, whereas the supports of $h_{p+1}, \ldots, h_k$ do not. Now g and f' have the same value, and hence

$$\lambda_1 + \ldots + \lambda_p \;=\; w(h) \;=\; w(g) - w(f) \;=\; w(f') - w(f) \;=\; \delta.$$

Moreover, $\gamma'(h_i) \geq \gamma'(f_C)$ for $i = 1, \ldots, p$, since C is a cycle of minimal cost containing r. Finally, $\gamma'(h_i) \geq 0$ for $i = p+1, \ldots, k$, because there are no directed cycles of negative cost with respect to γ'. Thus

$$\begin{aligned} \gamma'(g) - \gamma'(f) \;=\; \gamma'(h) \;&=\; \lambda_1\gamma'(h_1) + \ldots + \lambda_k\gamma'(h_k) \\ &\geq\; (\lambda_1 + \ldots + \lambda_p)\gamma'(f_C) \\ &=\; \delta\gamma'(f_C) \;=\; \gamma'(f') - \gamma'(f), \end{aligned}$$

which yields $\gamma'(g) \geq \gamma'(f')$, as desired. Thus we have established the following fundamental result.

Theorem 10.5.1. *Let $N = (G, c, s, t)$ be a flow network with cost function γ, and suppose that there are no directed cycles of negative cost with respect to γ. Moreover, let f be an optimal flow of value w on N, and P an augmenting path of minimal cost in the auxiliary network $N' = (G', c', s, t)$ with respect to the cost function γ' defined above. If f' is the flow which results from augmenting f along P (with capacity δ), then f' is an optimal flow of value $w + \delta$. If c, f, and γ are integral, then so is f'.* □

Theorem 10.5.1 shows that the following algorithm is correct. As the algorithm is merely a variation of the algorithm of Ford and Fulkerson, we only give an informal description; the reader should have no difficulties in writing out a detailed version.

Algorithm 10.5.2. Let $N = (G, c, s, t)$ be a flow network, where the capacity function c is integral, and let γ be a cost function such that G does not contain any directed cycles having negative cost. The algorithm constructs an integral

optimal flow of value v on N (if possible).

Procedure OPTFLOW$(G, c, s, t, \gamma, v; f, \text{sol})$

```
(1) for e ∈ E do f(e) ← 0 od
(2) sol ← true, val ← 0;
(3) while sol = true and val < v do
(4)       construct the auxiliary network N' = (G', c', s, t) with
          cost function γ';
(5)       if t is not accessible from s in G'
(6)       then sol ← false
(7)       else determine a shortest path P from s to t in (G', γ');
(8)            δ ← min{c'(e): e ∈ P}; δ' ← min(δ, v − val); val ← val + δ';
(9)            augment f along P by δ'
(10)      fi
(11) od
```

The Boolean variable *sol* indicates whether the problem has a solution (that is, whether there exists a flow of value v on N). If *sol* has value *true* at the end of the algorithm, then f is an optimal flow of value v.

Note that at most v iterations of the **while**-loop are needed: the value val of the flow is increased by at least 1 during each iteration. Constructing N' and augmenting f needs $O(|E|)$ steps in each iteration. A shortest path with respect to γ may be determined with complexity $O(|V|^3)$ using the algorithm of Floyd and Warshall of Section 3.9. Then we get a (non-polynomial) overall complexity of $O(|V|^3 v)$ for Algorithm 10.5.2.

If we assume that γ is nonnegative, we may use Dijkstra's algorithm instead of the algorithm of Floyd and Warshall for determining a shortest path from s to t in (G', γ') during the first iteration; this takes only $O(|V|^2)$ steps. However, during the following iterations, negative values of γ' always occur, namely for backward edges. We shall now describe a trick due to Edmonds and Karp [EdKa72] which allows us to use Dijkstra's algorithm in spite of the negative values of γ': we replace γ' by an appropriate nonnegative auxiliary function γ^*.

Let f be an optimal flow on N of value w. Suppose that we have already determined an augmenting path P of shortest length from s to t in (G', γ'), and also all distances $d'(s, x)$ in (G', γ'). As mentioned above, this is possible with complexity $O(|V|^2)$ for $f = 0$ if we use Dijkstra's algorithm. Let us denote the augmented optimal flow (obtained from P) by f'; the auxiliary network corresponding to f', by $N'' = (G'', c'', s, t)$ (this differs from our notation of Chapter 6); and the new cost function, by γ''. We require a shortest path P' from s to t in (G'', γ''), and also all distances $d''(s, x)$ in (G'', γ''). Now we replace $\gamma''(e)$ for each edge $e = uv$ of G'' by $\gamma^*(e)$, where

$$\gamma^*(e) \;=\; \gamma''(e) + d'(s, u) - d'(s, v), \tag{10.7}$$

and denote the distances in (G'', γ^*) by $d^*(s, x)$. Note that (10.7) implies

$$\gamma^*(X) \;=\; \gamma''(X) - d'(s,x)$$

for every path X from s to x in G''; in particular, a shortest path from s to x in G'' with respect to γ^* is also a shortest path with respect to γ''. Thus the distances

$$d''(s,x) \;=\; d^*(s,x) + d'(s,x)$$

with respect to γ'' can be calculated easily from those with respect to γ^*. Hence we may use the function γ^* in the algorithm instead of γ''. To see that γ^* is indeed nonnegative, consider an arbitrary edge $e = uv$ of G''. If e is not contained in the augmenting path P used for constructing f' or if e is a forward edge of P, then e is an edge of G' as well. In this case, $\gamma''(e) = \gamma'(e)$ and hence $d'(s,u)+\gamma'(e) \geq d'(s,v)$, by definition of the distance; thus $\gamma^*(e) \geq 0$. And if $e = uv$ is a backward edge in P, then $e' = vu$ is an edge of G' and $d'(s,u) = d'(s,v) + \gamma'(e')$, since P is a path of shortest length with respect to γ'. Now $\gamma''(e) = -\gamma'(e')$ shows $\gamma^*(e) = 0$ in this case.

Hence we may use Dijkstra's algorithm for (G'', γ^*) and determine the distances and a shortest augmenting path P' with complexity $O(|V|^2)$ or $O(|E| \log |V|)$; see Theorems 3.7.2 and 3.7.7. We have proved the following result [EdKa72]:

Theorem 10.5.3. *Let $N = (G, c, s, t)$ be a flow network with integral capacity function c and nonnegative cost function γ. Then Algorithm 10.5.2 can be used to determine an optimal flow of value v with complexity $O(v|V|^2)$ or $O(v|E| \log |V|)$.* □

In Section 12.5, we will apply Algorithm 10.5.2 to an important class of examples. A particular advantage of the algorithm is that it allows us to construct optimal flows for all possible values recursively. We denote the cost of an optimal flow of value v on N by $\gamma(v)$. Then Algorithm 10.5.2 may be used to find the *cost curve* of N: the function $v \mapsto \gamma(v)$.

Exercise 10.5.4. Discuss the properties of the cost curve of $N = (G, c, s, t)$, where the cost function γ is nonnegative and the capacity function c is integral. In particular, show that the cost curve is a *convex* function:

$$\gamma(\lambda v + (1-\lambda)v') \;\leq\; \lambda\gamma(v) + (1-\lambda)\gamma(v')$$

for all v, v' and all λ with $0 \leq \lambda \leq 1$.

Exercise 10.5.5. Discuss the complexity of the assignment problem introduced in Example 10.1.4.

10.6 Potentials and ε-optimality

This section provides the necessary foundation for the polynomial algorithms of Goldberg and Tarjan [GoTa90] for determining optimal circulations (which

will be described in the subsequent sections). Similar to Section 6.6, we begin by introducing a different presentation of circulations which will result in some technical simplifications; see the footnote to Theorem 10.6.4. As it is not quite as obvious as it was for flow networks that the new notation is indeed equivalent to the original definitions, we shall treat the necessary transformations in detail.

Construction 10.6.1. Let $G = (V, E)$ be a digraph with capacity constraints b and c. Our first step is to replace each pair of antiparallel edges by a single edge (having either of the two possible orientations). Thus let $e' = uv$ and $e'' = vu$ be any two antiparallel edges in G. We replace e' and e'' by the edge $e = uv$ with capacity constraints

$$b(e) = b(e') - c(e'') \qquad \text{and} \qquad c(e) = c(e') - b(e'').$$

This definition makes sense: $b(e') \leq c(e')$ and $b(e'') \leq c(e'')$ immediately imply $b(e) \leq c(e)$. If f is a feasible circulation for $N = (G, b, c)$, then f remains feasible after the above transformation of N if we put $f(e) = f(e') - f(e'')$. Conversely, let f be a feasible circulation on the transformed network N'. We need to consider what happens to a new edge e; as f is feasible,

$$b(e') - c(e'') \;\leq\; f(e) \;\leq\; c(e') - b(e''). \tag{10.8}$$

We now have to distribute $f(e)$ into two parts $f(e')$ and $f(e'')$ so that f is also feasible in the original network. Thus we look for values x and y satisfying

$$f(e) = x - y; \quad b(e') \leq x \leq c(e'); \quad \text{and} \quad b(e'') \leq y \leq c(e''),$$

which is equivalent to

$$\max\{b(e'), b(e'') + f(e)\} \;\leq\; x \;\leq\; \min\{c(e'), c(e'') + f(e)\}.$$

It is easy to see that it is indeed possible to choose x appropriately; this follows immediately from (10.8).[6]

Thus we may now assume that $N = (G, b, c)$ does not contain any pair of antiparallel edges. In our second step, we *symmetrize* G and f by reintroducing antiparallel edges: for each edge $e = uv$, we add the antiparallel edge $e' = vu$ and define b, c, and f as follows:

$$b(e') = -c(e), \qquad c(e') = -b(e), \qquad f(e') = -f(e).$$

In this way, f becomes a feasible circulation for the new symmetric network, since $b(e) \leq f(e) \leq c(e)$ implies $-c(e) \leq -f(e) \leq -b(e)$. Note that it is not necessary to state lower bounds explicitly any more: the lower bound $b(e) \leq f(e)$ follows from $-f(e) = f(e') \leq c(e') = -b(e)$.

[6] A similar argument shows that we do not need parallel edges; indeed, we have always excluded parallel edges in our study of flows and circulations.

For convenience, we consider f also as a function from $V \times V$ to $\mathbb{R}$: we do not distinguish between $f(e)$ and $f(u,v)$, where $e = uv$ is an edge of G; and we put $f(u,v) = 0$ whenever uv is not an edge of G. We proceed similarly for c. Then the compatibility condition (Z2) for f is replaced by the condition

$$-f(v,u) = f(u,v) \leq c(u,v) \qquad \text{for all } (u,v) \in V \times V. \tag{10.9}$$

As in Section 6.6, we define

$$e(v) \;=\; e_f(v) \;=\; \sum_{u \in V} f(u,v);$$

again, the flow conservation condition (Z1) is now written as

$$e_f(v) = 0 \qquad \text{for all } v \in V. \tag{10.10}$$

From now on, we can restrict our attention to networks $N = (G,c)$, where $c\colon V \times V \to \mathbb{R}$ may also take negative values.[7] A (feasible) *circulation* on N is a mapping $f\colon V \times V \to \mathbb{R}$ satisfying conditions (10.9) and (10.10). A mapping which satisfies condition (10.9) only is called a *pseudoflow* on N. We still have to define a cost function $\gamma\colon V \times V \to \mathbb{R}$ to be able to consider optimality. Now the antisymmetry conditions for circulations force us to require that γ is likewise antisymmetric:[8]

$$\gamma(u,v) = -\gamma(v,u) \qquad \text{for all } (u,v) \in V \times V.$$

Then the cost of a pseudoflow f is defined as

$$\gamma(f) \;=\; \frac{1}{2} \sum_{(u,v)} \gamma(u,v) f(u,v).$$

The factor $1/2$ is introduced here since the cost of the flow is counted twice for each edge uv of the original digraph G in the above sum; note that $\gamma(u,v)f(u,v) = \gamma(v,u)f(v,u)$. A pseudoflow or circulation of minimal cost is called *optimal*. This finishes the transformation of our usual setup to the definition of circulations used in [GoTa90].[9]

[7] Of course, c is not completely arbitrary: we should have $-c(v,u) \leq c(u,v)$; otherwise, there are no feasible circulations on N.

[8] Note that the transformation we have described tacitly assumes that pairs of antiparallel edges in the original digraph have the same cost; otherwise, the elimination technique used makes no sense. While this assumption will be satisfied in many applications, we can use the following little trick if it should be violated: for each pair of antiparallel edges with different costs, we subdivide one of the edges into two edges and then assign to both new edges the same capacities and half the cost of the original edge.

[9] For an intuitive interpretation of a circulation in the new sense, consider only its positive part – which lives on a network without antiparallel edges.

As in Section 6.6, we now introduce a residual graph G_f with respect to a given pseudoflow f; if f is a circulation, this graph corresponds to the auxiliary network used in Section 10.4. Let us define the *residual capacity* $r_f : V \times V \to \mathbb{R}$ by

$$r_f(v,w) \;=\; c(v,w) - f(v,w) \quad \text{for all } (v,w) \in V \times V.$$

If $r_f(v,w) > 0$ for some edge vw, we may use this edge to move some extra flow: in our intuitive interpretation, vw is a non-saturated edge. Such an edge is called a *residual edge*. The *residual graph* with respect to f,

$$G_f = (V, E_f), \text{ where } E_f \;=\; \{(v,w) \in V \times V : r_f(v,w) > 0\},$$

corresponds to the auxiliary network introduced in the classical approach.

Exercise 10.6.2. Describe a procedure RESIDUAL for constructing the residual graph.

Next we want to establish a further optimality criterion for circulations. We need a definition and a lemma first. A *potential* or a *price function* on the vertex set V is just a mapping $p : V \to \mathbb{R}$. For a given potential p and a given cost function γ, the *reduced cost function* γ_p is defined by[10]

$$\gamma_p(u,v) \;=\; \gamma(u,v) + p(u) - p(v). \tag{10.11}$$

The following lemma is easily verified and will be left to the reader; the second part of this lemma is particularly important.

Lemma 10.6.3. *Let (G,c) be a network with cost function γ, and let p be a potential on V. Then one has*

$$\gamma_p(P) \;=\; \gamma(P) + p(u) - p(v)$$

for every directed path P in G with start vertex u and end vertex v. In particular, $\gamma_p(C) = \gamma(C)$ for every directed cycle C in G. □

Theorem 10.6.4. *Let $N = (G,c)$ be a network with cost function γ, and let f be a circulation on N. Then the following four statements are equivalent:*

(a) *f is optimal.*

(b) *The residual graph G_f does not contain any directed cycles of negative length (with respect to γ).*

(c) *There exists a potential p on V such that $\gamma_p(u,v) \geq 0$ for all $uv \in E_f$.*

[10] Note that the following transformation has the same form as the one used in equation (10.7).

(d) *There exists a potential p on V which satisfies the condition*

$$\gamma_p(u,v) < 0 \quad \Longrightarrow \quad f(u,v) = c(u,v)$$

for all $(u,v) \in V \times V$.[11]

Proof. Conditions (a) and (b) are equivalent by Lemma 10.4.2. Moreover, conditions (c) and (d) are obviously equivalent if we consider the definition of the residual graph G_f. Thus it is sufficient to show that conditions (b) and (c) are equivalent. First, let p be a potential satisfying condition (c). Then all cycles in G_f have nonnegative length with respect to the reduced cost γ_p. By Lemma 10.6.3, the analogous condition holds for γ; hence (b) is satisfied.

Conversely, suppose that condition (b) holds. We construct an auxiliary graph H_f by adding a new vertex s and all edges sv (with $v \in V$) to G_f. By construction, s is a root of H_f. We extend γ to H_f by putting $\gamma(sv) = 0$ for all $v \in V$. As G_f (and hence H_f) does not contain any cycles of negative length, Theorem 3.4.4 yields the existence of an SP-tree T with root s for H_f. We define a potential p by $p(v) = d_T(s,v)$, where $d_T(s,v)$ denotes the distance of s from v in the network (T,γ). Then, by Exercise 3.4.6,

$$d_T(s,v) \leq d_T(s,u) + \gamma(u,v)$$

for all edges uv of G_f; hence

$$\gamma_p(u,v) = \gamma(u,v) + p(u) - p(v) \geq 0 \quad \text{for all } uv \in E_f,$$

so that p satisfies condition (c). □

The basic idea of the algorithm of Goldberg and Tarjan is to construct a sequence of progressively improving circulations on (G,c). Theorem 10.6.4 suggests the following weakening of the notion of optimality: a circulation – and, more generally, a pseudoflow – f is called *ε-optimal* , where ε is a

[11] This optimality criterion for potentials is one example for how the different way of description used in this section simplifies the technical details of our presentation. Of course, an analogous criterion can be proved using the standard notation for a network (G,b,c). However, we would then need three conditions which have to be satisfied for all edges $uv \in E$:

$$\begin{aligned} f(u,v) = b(u,v) \quad &\Longrightarrow \quad c_p(u,v) \geq 0, \\ f(u,v) = c(u,v) \quad &\Longrightarrow \quad c_p(u,v) \leq 0, \\ b(u,v) < f(u,v) < c(u,v) \quad &\Longrightarrow \quad c_p(u,v) = 0; \end{aligned}$$

see [AhMO93], p. 330. These conditions are called *complementary slackness conditions*; they are a special case of the corresponding conditions used in linear programming. It may be checked that the potentials $p(v)$ correspond to the dual variables if the problem of determining an optimal circulation is written as a linear program.

nonnegative real number, if there exists a potential p on V satisfying the condition

$$\gamma_p(u,v) \;\geq\; -\varepsilon \quad \text{for all } uv \in E_f. \tag{10.12}$$

Obviously, (10.12) can also be written as

$$\gamma_p(u,v) < -\varepsilon \;\;\Longrightarrow\;\; uv \notin E_f \quad \text{for all } (u,v) \in V \times V. \tag{10.13}$$

By Theorem 10.6.4, 0-optimality is the same as optimality. The following simple result illustrates the importance of the notion of ε-optimality: at least for integral cost functions, it suffices to determine an almost optimal circulation.

Theorem 10.6.5. *Let $N = (G, c)$ be a network with an integral cost function $\gamma\colon V \times V \to \mathbb{Z}$. Moreover, let $\varepsilon > 0$ be a real number satisfying the condition $\varepsilon|V| < 1$. Then an ε-optimal circulation on N is already optimal.*

Proof. Let p be a potential satisfying condition (10.12), and let C be a directed cycle in G_f. Then Lemma 10.6.3 implies

$$\gamma(C) \;=\; \gamma_p(C) \;\geq\; -|C|\varepsilon \;\geq\; -|V|\varepsilon \;>\; -1.$$

But γ is integral, and hence $\gamma(C) \geq 0$. Thus (G_f, γ) does not contain any directed cycles of negative length, so that f is optimal by Theorem 10.6.4. □

We now need a method for checking whether a given circulation is ε-optimal and for constructing an associated potential satisfying condition (10.12). This can be done using an argument similar to the proof of Theorem 10.6.4; the corresponding result actually holds for pseudoflows in general.

Theorem 10.6.6. *Let f be a pseudoflow on the network $N = (G, c)$ with cost function γ. For $\varepsilon > 0$, we define the function $\gamma^{(\varepsilon)}$ by*

$$\gamma^{(\varepsilon)}(u,v) \;=\; \gamma(u,v) + \varepsilon \qquad \textit{for all } (u,v) \in V \times V.$$

Then f is ε-optimal if and only if the network $(G_f, \gamma^{(\varepsilon)})$ does not contain any directed cycles of negative length.

Proof. We define the graph H_f as in the proof of Theorem 10.6.4. Note that all directed cycles in H_f actually lie in G_f, because s has indegree 0. We extend $\gamma^{(\varepsilon)}$ to H_f by putting $\gamma^{(\varepsilon)}(sv) = 0$ for all $v \in V$; then we may check the criterion given in the assertion for H_f (instead of G_f). First suppose that f is ε-optimal with respect to the potential p, and let C be a directed cycle in G_f. Lemma 10.6.3 implies $\gamma(C) = \gamma_p(C) \geq -|C|\varepsilon$ and, hence, indeed

$$\gamma^{(\varepsilon)}(C) \;=\; \gamma(C) + |C|\varepsilon \;\geq\; 0.$$

Conversely, suppose that $(G_f, \gamma^{(\epsilon)})$ does not contain any directed cycles of negative length. By analogy to the proof of Theorem 10.6.4, we may choose

an SP-tree T for $(H_f, \gamma^{(\varepsilon)})$ and define a potential p by $p(v) = d_T(s, v)$. Then $\gamma_p^{(\varepsilon)}(u, v) \geq 0$ and thus $\gamma_p(u, v) \geq -\varepsilon$ for all $(u, v) \in G_f$. □

The proof of Theorem 10.6.6 shows the validity of the following corollary, which allows us, using Exercise 3.10.3, to construct with complexity $O(|V||E|)$ the associated potential p satisfying condition (10.12) for a given ε-optimal circulation.

Corollary 10.6.7. *Let f be an ε-optimal pseudoflow on $N = (G, c)$ with respect to the cost function γ. Moreover, let T be an SP-tree with root s in the auxiliary graph H_f with respect to $\gamma^{(\varepsilon)}$. Then the potential p defined by $p(v) = d_T(s, v)$ satisfies condition (10.12).* □

Exercise 10.6.8. Write down a procedure POTENTIAL explicitly which with complexity $O(|V||E|)$ determines a potential as in Corollary 10.6.7.

Remark 10.6.9. Note that every pseudoflow f – in particular, every circulation – on (G, c) is ε-optimal (with respect to γ) for some value of ε. For example, if C is the maximum of all values $|\gamma(u, v)|$ and if the potential p is chosen as the zero potential, f is trivially C-optimal.

Now the problem arises how we may determine the *smallest* ε such that a given pseudoflow f is still ε-optimal; in this case, we say that f is *ε-tight*. We need a further concept. Let (H, w) be a network. For every directed cycle C in H,

$$m(C) \;=\; \frac{w(C)}{|C|}$$

is called the *mean weight* of C.[12] Moreover,

$$\mu(H, w) \;=\; \min\{m(C) \colon C \text{ a directed cycle in } (H, w)\}$$

is called the *minimum cycle mean*.

Theorem 10.6.10. *Let f be a pseudoflow on (G, c) which is not optimal with respect to the cost function γ. Then f is ε-tight, where $\varepsilon = -\mu(G_f, \gamma)$.*

Proof. Let C be a directed cycle in G_f, and denote by ε the real number for which f is ε-tight. Then Theorem 10.6.6 implies (using the notation of Theorem 10.6.6)

$$\gamma^{(\varepsilon)}(C) \;=\; \gamma(C) + |C|\varepsilon \;\geq\; 0;$$

thus $m(C) = \gamma(C)/|C| \geq -\varepsilon$. As this holds for every directed cycle, we conclude $\mu := \mu(G_f, \gamma) \geq -\varepsilon$; hence $\varepsilon \geq -\mu$.

Conversely, every directed cycle C satisfies $m(C) = \gamma(C)/|C| \geq \mu$, by definition. Therefore

[12] In our context, the terms *mean cost* or *mean length* would make more sense; however, we do not want to deviate from common usage.

$$\gamma^{(-\mu)}(C) \;=\; \gamma(C) - |C|\mu \;\geq\; 0.$$

Again by Theorem 10.6.6, f is at least $(-\mu)$-optimal, so that also $\varepsilon \leq -\mu$. □

It remains to address the question how the minimum cycle mean can be determined efficiently. By a result of Karp [Kar78], this may be done with complexity $O(|V||E|)$ – the same complexity as for determining an SP-tree, or for checking whether any directed cycle of negative length exists (see 3.10.3); of course, determining $\mu(H, w)$ also answers the latter question. Karp's algorithm is based on the following characterization of $\mu(H, w)$.

Theorem 10.6.11. *Let (H, w) be a network on a digraph $H = (V, E)$, and suppose that H has a root s and contains directed cycles. For each vertex v and each positive integer k, let $F_k(v)$ denote the minimal length of a directed walk from s to v consisting of exactly k edges; if no such walk exists, we put $F_k(v) = \infty$. Then, with $n = |V|$,*

$$\mu(H, w) \;=\; \min_{v \in V} \max \left\{ \frac{F_n(v) - F_k(v)}{n - k} \colon\; k = 1, \ldots, n-1 \right\}. \tag{10.14}$$

Proof. We first prove the desired identity for the special case $\mu(H, w) = 0$. Then (H, w) does not contain any directed cycles of negative length, so that the shortest length of a path from s to v equals the shortest length of a walk from s to v. Therefore

$$F_n(v) \;\geq\; d(s, v) \;=\; \min \{F_k(v) \colon\; k = 1, \ldots, n-1\},$$

and thus

$$F_n(v) - d(s, v) \;=\; \max \{F_n(v) - F_k(v) \colon\; k = 1, \ldots, n-1\} \;\geq\; 0$$

and

$$\max \left\{ \frac{F_n(v) - F_k(v)}{n - k} \colon\; k = 1, \ldots, n-1 \right\} \;\geq\; 0.$$

Hence it suffices to prove the existence of some vertex v with $F_n(v) = d(s, v)$. Let C be any cycle of weight 0 and u a vertex in C; moreover, let P be a path of length $d(s, u)$ from s to u. Now we may append C to P any number of times to obtain a shortest walk W from s to u. Note that any part W' of W beginning in s and ending in v, say, has to be a shortest walk from s to v. Obviously, we may choose v in such a way that W' consists of exactly n edges; this vertex v satisfies $F_n(v) = d(s, v)$.

It remains to consider the case $\mu(H, w) = \mu \neq 0$. We replace the given weight function w with the function w' defined by

$$w'(uv) \;=\; w(uv) - \mu \qquad \text{for all } uv \in E.$$

Then every cycle C in H satisfies $w'(C) = w(C) - |C|\mu$, and therefore $m'(C) = m(C) - \mu$. In other words, replacing w by w' results in reducing the minimum

cycle mean by μ, so that $\mu(H, w') = 0$. But we have already established the assertion in this case, and therefore

$$\mu(H, w') \;=\; 0 \;=\; \min_{v \in V} \max \left\{ \frac{F'_n(v) - F'_k(v)}{n-k} \colon\; k = 1, \ldots, n-1 \right\}.$$

On the other hand, every walk W in H satisfies $w'(W) = w(W) - |W|\mu$, so that $F'_l(v) = F_l(v) - l\mu$. This implies

$$\frac{F'_n(v) - F'_k(v)}{n-k} \;=\; \frac{(F_n(v) - n\mu) - (F_k(v) - k\mu)}{n-k} \;=\; \frac{F_n(v) - F_k(v)}{n-k} - \mu,$$

and the assertion follows. □

Corollary 10.6.12. *Let $H = (V, E)$ be a connected digraph with weight function $w \colon E \to \mathbb{R}$. Then $\mu(H, w)$ can be determined with complexity $O(|V||E|)$.*

Proof. By Theorem 2.6.6, we may check with complexity $O(|E|)$ whether H is acyclic; in this case, $\mu(H, w) = \infty$. Otherwise we may, if necessary, add a root s to H (as we did in the proof of Theorem 10.6.4) without introducing any new directed cycles. Then Theorem 10.6.11 may be applied to the new graph. The values $F_k(v)$ can be calculated recursively using the initial values

$$F_0(s) = 0, \quad F_0(v) = \infty \quad \text{for } v \neq s,$$

and the identity

$$F_k(v) \;=\; \min\{F_{k-1}(u) + w(uv) \colon\; uv \in E\};$$

this obviously takes $O(|V||E|)$ steps. After we have calculated all the $F_k(v)$, we may determine $\mu(H, w)$ with $O(|V|^2)$ comparisons, by Theorem 10.6.10. As H is connected, $|V|$ is dominated by $|E|$, and the assertion follows. □

Exercise 10.6.13. Write down a procedure MEANCYCLE having the properties described in Corollary 10.6.12. In addition, your procedure should also construct a cycle which has the minimum cycle mean as its mean weight.

There is also an algorithm with complexity $O(|V|^{1/2}|E| \log(|V|C))$ for determining $\mu(H, w)$, where C is the maximum of the absolute values $|\gamma(u, v)|$; see [OrAh92]. Another efficient algorithm is in [YoTO91]; experiments with random graphs suggest that its average complexity is $O(|E| + |V| \log |V|)$.

Let us summarize the preceding results as follows:

Theorem 10.6.14. *Let f be a circulation on a network $N = (G, c)$ with cost function γ. Then the number ε for which f is ε-tight can be determined with complexity $O(|V||E|)$.*

Proof. We calculate $\mu = \mu(G_f, \gamma)$; this can be done with the desired complexity by Corollary 10.6.12. If $\mu \geq 0$, G_f does not contain any directed cycles of negative length with respect to γ, so that f is optimal by Theorem 10.6.4, and hence $\varepsilon = 0$. Otherwise $\mu < 0$ and f is not optimal. But then $\varepsilon = -\mu$ by Theorem 10.6.10. □

Theorem 10.6.14 allows us to determine an optimal measure for the quality of any given circulation on N. As hinted before, the algorithm of Goldberg and Tarjan is based on finding a sequence of ε-optimal circulations for decreasing ε and finally applying Theorem 10.6.5 (in the integral case). We will present their algorithm in the next section.

Exercise 10.6.15. Write down a procedure TIGHT explicitly which determines the number ε of Theorem 10.6.14 with complexity $O(|V||E|)$.

10.7 Optimal circulations by successive approximation

In this section, we present a generic version of the polynomial algorithm of Goldberg and Tarjan [GoTa90] for determining optimal circulations; this rests on the ideas treated in the previous section. For the time being, we shall assume that we have already designed an auxiliary procedure REFINE which constructs from a given ε-optimal circulation f with associated potential p an ε'-optimal circulation f' and a corresponding potential p', where $\varepsilon' = \varepsilon/2$; an efficient version of REFINE will be derived in the next section. We always assume that the network under consideration does not have any antiparallel edges; we may do so in view of Construction 10.6.1.

Algorithm 10.7.1. Let $N = (G_0, b, c)$ be a network with cost function γ, where $G_0 = (V, E_0)$ is a digraph without any pairs of antiparallel edges. The algorithm constructs an optimal circulation f_0 on N, or determines the nonexistence of feasible solutions.

Procedure OPTCIRC$(G_0, b, c, \gamma; \text{legal}, f_0)$

```
(1) LEGCIRC(G_0, b, c; legal, f)
(2) if legal = true then
(3)     E ← E_0;
(4)     for uv ∈ E_0 do
(5)         E ← E ∪ {vu}; f(v,u) ← −f(u,v);
(6)         γ(v,u) ← −γ(u,v); c(v,u) ← −b(u,v)
(7)     od
(8)     G ← (V,E);
(9)     TIGHT (G, c, γ, f; ε)
(10)    while ε > 0 do
(11)        POTENTIAL(G, c, γ, f, ε; p);
(12)        REFINE(G, c, γ, f, ε, p; f);
```

(13) TIGHT$(G, c, \gamma, f; \varepsilon)$
(14) **od**
(15) $f_0 \leftarrow f|E_0$;
(16) **fi**

Theorem 10.7.2. *Let $N = (G_0, b, c)$ be a network with an integral cost function γ, where $G_0 = (V, E_0)$ is a digraph without any pairs of antiparallel edges, and assume the existence of feasible circulations. Suppose that* REFINE *is a procedure which constructs from an ε-optimal circulation f with associated potential p an $\varepsilon/2$-optimal circulation and a corresponding potential. Then Algorithm 10.7.1 determines with complexity $O(\log(|V|C))$ an optimal circulation f_0 on N, where $C = \max\{|\gamma(e)| : e \in E_0\}$.*

Proof. By Corollary 10.2.5, step (1) of the algorithm constructs a feasible circulation f on N. Steps (3) to (8) determines the symmetrized form (G, c) of the network (G_0, b, c) as well as corresponding versions of the functions f and γ, as in Construction 10.6.1. In step (9), the procedure TIGHT calculates the value of ε for which f is ε-tight. If $\varepsilon > 0$ – so that f is not yet optimal – the algorithm constructs an associated potential p for f; changes f to an $\varepsilon/2$-optimal circulation again denoted by f; and determines the precise value of ε for which the new f is ε-tight. All this happens during the **while**-loop (10) to (14); note that this **while**-loop terminates only if $\varepsilon = 0$ so that the current f is an optimal circulation. As ε is decreased with each iteration of the loop by at least a factor of $1/2$ (note that ε may actually be smaller than this bound guarantees!), and as the initial circulation is C-optimal by Remark 10.6.9, an ε-optimal circulation with $\varepsilon < 1/|V|$ is reached after at most $O(\log(|V|C))$ iterations. By Theorem 10.6.5, this circulation is already optimal – so that actually $\varepsilon = 0$, and the **while**-loop is terminated. Finally, in step (15), f_0 is assigned the values of the final optimal circulation on (G, c), restricted to the original network. □

In the remainder of this section, we show that Algorithm 10.7.1 terminates after at most $O(|E| \log |V|)$ iterations, even if γ is not integral. This requires some more work; we begin by showing that the flow value $f(e)$ on an edge e cannot be further changed in subsequent iterations provided that the reduced cost of e is sufficiently large.

Theorem 10.7.3. *Let f be an ε-optimal circulation with associated potential p on a network $N = (G, c)$ with cost function γ, and assume $\varepsilon > 0$. Moreover, suppose*

$$|\gamma_p(u, v)| \;\geq\; |V|(\varepsilon + \delta)$$

for some edge uv and some $\delta \geq 0$. Then every δ-optimal circulation g satisfies $g(u, v) = f(u, v)$.

Proof. Because of the antisymmetry of f and γ, we may assume $\gamma_p(u, v) \geq 0$. Now let g be any circulation with $g(u, v) \neq f(u, v)$. Our hypothesis implies

$\gamma_p(u,v) > \varepsilon$, and hence $\gamma_p(v,u) < -\varepsilon$; thus, by (10.13), $vu \notin E_f$. Using this, we obtain

$$f(u,v) \;=\; -f(v,u) \;=\; -c(v,u) \;\leq\; -g(v,u) \;=\; g(u,v).$$

In view of $g(u,v) \neq f(u,v)$, we conclude $g(u,v) > f(u,v)$. We now show that g cannot be δ-optimal. For this purpose, we consider the digraph $G^>$ with vertex set V and edge set

$$E^> \;=\; \{xy \in E\colon g(x,y) > f(x,y)\}.$$

Obviously, $G^>$ is a subdigraph of G_f containing the edge uv. We show first that $G^>$ contains a directed cycle through uv. Consider the digraph H whose edges are all the edges $e \in E_f$ satisfying $h(e) := g(e) - f(e) \neq 0$. We color the edges of H either black or green, depending on whether $h(e) > 0$ or $h(e) < 0$. By the antisymmetry of f and g, an edge $e = xy$ is black if and only if the antiparallel edge $e' = yx$ is green. By the painting lemma (Theorem 10.3.9), there exists either a cycle K or a cocycle C containing $e_0 = uv$ so that all its black edges have the same orientation as e_0, whereas all its green edges are oriented in the opposite direction. In the first case, we may replace all green edges occurring in K by their corresponding antiparallel edges, so that we get a directed cycle in $G^>$ containing e_0. The second case leads to a contradiction. To see this, let (S,T) be the cut of H corresponding to C. By Exercise 10.3.4, $h(S,T) = h(T,S)$. However, the properties of C given in the painting lemma together with $e_0 \in C$ imply $h(S,T) > 0$ and $h(T,S) < 0$. Thus this case cannot occur.

Thus $G^>$ indeed contains a directed cycle K through uv; note that all edges of K are in E_f. Using Lemma 10.6.3 and the definition of ε-optimality,

$$\begin{aligned}\gamma(K) \;&=\; \gamma_p(K) \;\geq\; \gamma_p(u,v) - (|K|-1)\varepsilon\\ &\geq\; |V|(\varepsilon+\delta) - (|V|-1)\varepsilon\\ &>\; |V|\delta \;\geq\; |K|\delta.\end{aligned}$$

Now let $\overline{K}$ be the cycle of G which we obtain by inverting the orientation of all edges of K. Then $\overline{K}$ is contained in $G^<$, where $G^<$ is the subdigraph with edge set

$$E^< \;=\; \{xy \in E : g(x,y) < f(x,y)\},$$

so that $G^<$ is likewise a subgraph of the residual graph G_g. The antisymmetry of γ implies

$$\gamma(\overline{K}) \;=\; -\gamma(K) \;<\; -|K|\delta \;=\; -|\overline{K}|\delta$$

and hence

$$\gamma^{(\delta)}(\overline{K}) \;=\; \gamma(\overline{K}) + \delta|\overline{K}| \;<\; 0.$$

By Theorem 10.6.6, g cannot be δ-optimal. □

Let us call an edge uv *ε-fixed* if the value $f(u,v)$ is the same for all ε-optimal circulations f on (G,c) with respect to γ.

Corollary 10.7.4. *Let f be an ε-optimal circulation with associated potential p on (G,c) with respect to the cost function γ, where $\varepsilon > 0$. Then every edge uv with $|\gamma_p(u,v)| \geq 2|V|\varepsilon$ is ε-fixed.* □

Lemma 10.7.5. *Let f be an ε-tight circulation with $\varepsilon \neq 0$ on the network $N = (G,c)$ with respect to the cost function γ, and let p be a corresponding potential. Moreover, let C be a directed cycle of minimum cycle mean in the residual graph G_f. Then $\gamma_p(u,v) = -\varepsilon$ for all $uv \in C$.*

Proof. By hypothesis,

$$\gamma_p(u,v) \;\geq\; -\varepsilon \quad \text{for all } uv \in E_f. \tag{10.15}$$

On the other hand, $\mu(G_f,\gamma) = -\varepsilon$ by Theorem 10.6.10; note that this number is negative because $\varepsilon \neq 0$. Then, by Lemma 10.6.3,

$$\frac{1}{|C|}\sum_{uv\in C}\gamma_p(u,v) \;=\; \frac{1}{|C|}\sum_{uv\in C}\gamma(u,v) \;=\; m(C) \;=\; -\varepsilon.$$

Now (10.15) implies the assertion. □

Lemma 10.7.6. *Let $N = (G,c)$ be a network with cost function γ, and denote by F_ε the set of all ε-fixed edges in G, where $\varepsilon > 0$. Also, assume the existence of an ε-tight circulation f. Then the set F_ε is a proper subset of F_δ for every $\delta \geq 0$ with $2\delta|V| \leq \varepsilon$.*

Proof. Trivially, $F_\varepsilon \subseteq F_\delta$. Thus we have to find some edge which is δ-fixed but not ε-fixed. As f is an ε-tight circulation, there exists a directed cycle C in the residual graph G_f of mean weight $m(C) = -\varepsilon$ (with respect to γ), by Theorem 10.6.10. Then we may increase f along C by a sufficiently small amount and get a new feasible circulation f'. We shall show that f' is likewise ε-optimal, so that the edges of C cannot be contained in F_ε.

Thus let p be a potential corresponding to f. Then $\gamma_p(u,v) \geq -\varepsilon$ for all edges $uv \in E_f$. The only edges $uv \in E_{f'}$ which are not necessarily contained in E_f as well are those edges for which the antiparallel edge vu lies in C; note that the cycle having opposite orientation to C indeed has to be contained in $G_{f'}$. Because of $vu \in E_f$ and by Lemma 10.7.5, these edges satisfy

$$\gamma_p(u,v) \;=\; -\gamma_p(v,u) \;=\; \varepsilon \;>\; 0,$$

so that f' is ε-optimal with respect to the same potential p.

Next we show that at least one edge of C is contained in F_δ. Let g be any δ-optimal circulation with associated potential p'. By the choice of C,

$$\gamma_{p'}(C) \;=\; \gamma_p(C) \;=\; \gamma(C) \;=\; -|C|\varepsilon,$$

where we have used Lemma 10.6.3 again. Therefore C has to contain an edge uv with

$$\gamma_{p'}(u,v) \;\leq\; -\varepsilon \;\leq\; -2|V|\delta.$$

Thus $|\gamma_{p'}(u,v)| \geq 2|V|\delta$; by Corollary 10.7.4, uv is contained in F_δ. □

Theorem 10.7.7. *If N admits feasible circulations, Algorithm 10.7.1 determines an optimal circulation on N in $O(|E|\log|V|)$ iterations of the* **while***-loop, under the assumption that* REFINE *satisfies the requirements of Theorem 10.7.2.*

Proof. Let f be an ε-optimal circulation calculated at some point of the algorithm. By our assumptions regarding REFINE, we need at most $O(\log|V|)$ iterations to construct a δ-tight circulation f' from f for some δ with $\delta \leq \varepsilon/2|V|$. If $\delta = 0$, the algorithm terminates. Otherwise, the set F_δ of δ-fixed edges contains at least one more edge than F_ε. Now the algorithm has to terminate for sure if *all* edges are δ-fixed, which takes at most $O(|E|\log|V|)$ iterations. □

Note that Algorithm 10.7.1 usually terminates earlier, since in most cases not all edges are 0-fixed: it is very well possible that there are several different optimal circulations. In the next section, we will show that the auxiliary procedure REFINE can be performed in $O(|V|^3)$ steps. The above results then yield the following theorem [GoTa90].

Theorem 10.7.8. *Let $N = (G, b, c)$ be a network with cost function γ which admits feasible circulations. Then Algorithm 10.7.1 determines with complexity $O(|E||V|^3\log|V|)$ an optimal circulation on N. If the cost function γ is integral, the complexity is also bounded by $O(|V|^3\log(|V|C))$, where $C = \max\{|\gamma(u,v)| : uv \in E\}$.* □

If G is not a dense graph, the complexity may be improved by using more intricate data structures; in particular, there is a version of REFINE which needs only $O(|V||E|\log(|V|^2/|E|))$ steps; see [GoTa90].

10.8 A polynomial procedure REFINE

We still need to fill the gap left in the last section and provide an auxiliary procedure REFINE with complexity $O(|V|^3)$. We shall present the procedure of Goldberg and Tarjan [GoTa90], which is quite similar to Algorithm 6.6.1 for determining a maximal flow on a flow network, even as far as the proofs are concerned. As in Section 6.6, we first give a generic version where the auxiliary operations used can be chosen in an arbitrary order. Afterwards, an appropriate way of choosing these operations will lead to a rather good complexity bound. Again, we call a vertex v *active* if its flow excess with respect to f satisfies the condition $e_f(v) > 0$.

Algorithm 10.8.1. Let (G, c) be a network as described in Construction 10.6.1 with cost function γ. Moreover, let f be an ε-optimal circulation with corresponding potential p. The algorithm determines an $\varepsilon/2$-optimal circulation and the corresponding potential.

Procedure REFINE$(G, c, \gamma, f, \varepsilon, p; f)$

(1) $\varepsilon \leftarrow \varepsilon/2$;
(2) **for** $uv \in E$ **do**
(3) $\quad \gamma_p(u,v) \leftarrow \gamma(u,v) + p(u) - p(v)$;
(4) $\quad$ **if** $\gamma_p(u,v) < 0$
(5) $\quad$ **then** $f(u,v) \leftarrow c(u,v)$; $f(v,u) \leftarrow -c(u,v)$;
(6) $\quad\quad r_f(u,v) \leftarrow 0$; $r_f(v,u) \leftarrow c(v,u) - f(v,u)$
(7) $\quad$ **fi**
(8) **od**
(9) **for** $v \in V$ **do** $e(v) \leftarrow \sum_u f(u,v)$ **od**
(10) **while** there exist admissible operations **do**
(11) $\quad$ choose some admissible operation and execute it
(12) **od**

Here the possible admissible operations are:

Procedure PUSH$(f, v, w; f)$

(1) $\delta \leftarrow \min(e(v), r_f(v,w))$;
(2) $f(v,w) \leftarrow f(v,w) + \delta$; $f(w,v) \leftarrow f(w,v) - \delta$;
(3) $r_f(v,w) \leftarrow r_f(v,w) - \delta$; $r_f(w,v) \leftarrow r_f(w,v) + \delta$;
(4) $e(v) \leftarrow e(v) - \delta$; $e(w) \leftarrow e(w) + \delta$.

The operation PUSH$(f, v, w; f)$ is *admissible* if v is active, $r_f(v,w) > 0$, and $\gamma_p(v,w) < 0$.

Procedure RELABEL$(f, v, p; f, p)$

(1) $\Delta \leftarrow \varepsilon + \min\{\gamma_p(v,w) \colon r_f(v,w) > 0\}$;
(2) $p(v) \leftarrow p(v) - \Delta$;
(3) **for** $w \in V \setminus \{v\}$ **do**
(4) $\quad \gamma_p(v,w) \leftarrow \gamma_p(v,w) - \Delta$; $\gamma_p(w,v) \leftarrow -\gamma_p(v,w)$
(5) **od**

The operation RELABEL$(f, v, p; f, p)$ is *admissible* if v is active and if $\gamma_p(v,w) \geq 0$ holds whenever $r_f(v,w) > 0$. Alternatively, we could describe the modification of the value $p(v)$ in RELABEL by the command

$$p(v) \leftarrow \max\{p(w) - \gamma(v,w) - \varepsilon \colon r_f(v,w) > 0\},$$

as in the original paper.

As in Section 6.6, we first prove that Algorithm 10.8.1 is correct, provided that it terminates. The following lemma is similar to Lemma 6.6.2 and equally obvious.

Lemma 10.8.2. *Let f be an ε-optimal pseudoflow on (G, c) with respect to the cost function γ, and let p be a corresponding potential. Moreover, let v be an active vertex. Then either* RELABEL(v) *is admissible, or there is an edge vw for which* PUSH(v, w) *is admissible.* □

Lemma 10.8.3. *Let f be an ε-optimal pseudoflow on (G, c) with respect to the cost function γ, and let p be a corresponding potential. Moreover, let v be an active vertex. Then the new pseudoflow which is obtained from a* PUSH*-operation on some edge vw is still ε-optimal. A* RELABEL(v)*-operation decreases $p(v)$ by at least ε; again, the pseudoflow remains ε-optimal after the* RELABEL*-operation.*

Proof. To prove the first claim, note that a PUSH(v, w) does not change the reduced cost of edges which already occur in G_f. If the edge wv is added to G_f by the PUSH(v, w), the conditions for the admissibility of a PUSH-operation yield $\gamma_p(v, w) < 0$, so that $\gamma_p(w, v) > 0$; hence the new residual edge wv satisfies the condition for ε-optimality.

Now consider a RELABEL(v)-operation. If this operation is admissible, we must have $\gamma_p(v, w) \geq 0$ for all $vw \in E_f$, so that

$$p(v) \;\geq\; p(w) - \gamma(v, w) \quad \text{for all } vw \in E_f$$

holds before RELABEL is performed. This implies

$$p'(v) \;=\; \max\{p(w) - \gamma(v, w) - \varepsilon \colon vw \in E_f\} \;\leq\; p(v) - \varepsilon$$

for the value $p'(v)$ of the potential after the RELABEL operation. Therefore $p(v)$ is decreased by at least ε during the RELABEL(v)-operation. The only edges whose reduced cost is changed by a RELABEL(v) are the edges which are incident with v. For every edge of the form wv, $\gamma_p(w, v)$ is increased by at least ε; trivially, this does not change the ε-optimality. Now consider a residual edge of the form vw. By definition of $p'(v)$, such an edge satisfies

$$p'(v) \;\geq\; p(w) - \gamma(v, w) - \varepsilon$$

and hence

$$\gamma_{p'}(v, w) \;=\; \gamma(v, w) + p'(v) - p(w) \;\geq\; -\varepsilon,$$

so that the condition for ε-optimality holds also in this case. □

Theorem 10.8.4. *Assume that Algorithm 10.8.1 terminates. Then the final pseudoflow f is an $\varepsilon/2$-optimal circulation.*

Proof. Note that the pseudoflow f constructed during the initialization phase (2) to (8) is actually 0-optimal (as all edges with negative reduced cost are saturated), so that it is for sure $\varepsilon/2$-optimal. Now Lemma 10.8.3 shows that the pseudoflow remains $\varepsilon/2$-optimal throughout the algorithm. By Lemma 10.8.2, the algorithm terminates only when there is no longer any active vertex. But this means $e(v) \leq 0$ for all vertices v; hence

$$\sum_v e(v) \;=\; \sum_{u,v} f(u, v) \;=\; 0$$

shows $e(v) = 0$ for all v. Thus the $\varepsilon/2$-optimal pseudoflow constructed during the last iteration of the algorithm is indeed a circulation. □

In order to show that Algorithm 10.8.1 terminates, we have to find an upper bound for the number of admissible operations executed during the algorithm. As in Section 6.6, we distinguish *saturating* PUSH-operations, namely those with $\delta = r_f(v,w)$, from *non-saturating* PUSH-operations. We begin by analyzing the RELABEL-operations. The following important lemma is analogous to Lemma 6.6.7.

Lemma 10.8.5. *Let f be a pseudoflow and g a circulation on (G,c). For each vertex v with $e_f(v) > 0$, there exist a vertex w with $e_f(w) < 0$ and a sequence of distinct vertices $v = v_0, v_1, \ldots, v_{k-1}, v_k = w$ with $v_i v_{i+1} \in E_f$ and $v_{i+1} v_i \in E_g$ for $i = 0, \ldots, k-1$.*

Proof. We define the directed graphs $G^>$ and $G^<$ as in the proof of Theorem 10.7.3; then $G^>$ is a subdigraph of G_f, and $G^<$ is a subdigraph of G_g. Moreover, $xy \in E^>$ if and only if $yx \in E^<$, since pseudoflows are antisymmetric. Hence it suffices to show the existence of a directed path

$$P: \quad v_0 = v \text{ — } v_1 \text{ — } \ldots \text{ — } v_k = w$$

with $e_f(w) < 0$ in $G^>$. Denote the set of vertices which are accessible from v in $G^>$ by S, and put $\overline{S} := V \setminus S$. (The set $\overline{S}$ might be empty.) For each pair (x,y) of vertices with $x \in S$ and $y \in \overline{S}$, we have $g(x,y) \leq f(x,y)$ by definition. As g is a circulation and as f and g are antisymmetric,

$$\begin{aligned} 0 &= \sum_{y \in \overline{S}} e_g(y) = \sum_{x \in V, y \in \overline{S}} g(x,y) \\ &= \sum_{x \in S, y \in \overline{S}} g(x,y) \leq \sum_{x \in S, y \in \overline{S}} f(x,y) \\ &= \sum_{x \in S, y \in V} f(x,y) = -\sum_{x \in S, y \in V} f(y,x) = -\sum_{x \in S} e_f(x). \end{aligned}$$

However, $v \in S$ and $e_f(v) > 0$. Therefore S has to contain a vertex w with $e_f(w) < 0$, proving the assertion. □

Lemma 10.8.6. *For each vertex v, at most $3|V|$ RELABEL(v)-operations are performed during Algorithm 10.8.1. Thus there are altogether at most $O(|V|^2)$ RELABEL-operations during the course of the algorithm.*

Proof. Note that the values of the potential can only decrease during the execution of REFINE, by Lemma 10.8.3. Now consider the situation immediately after some RELABEL(v)-operation, and let f be the ε-optimal pseudoflow with associated potential p at this point of time. Then $e_f(v) > 0$. In what follows, we denote the original ε-optimal circulation and the corresponding

potential – the input parameters of REFINE – by g and q. By Lemma 10.8.5, there exist a vertex w with $e_f(w) < 0$ and a directed path

$$P\colon\ v = v_0 \text{ — } v_1 \text{ — } \ldots \text{ — } v_{k-1} \text{ — } v_k = w$$

with $v_i v_{i+1} \in E_f$ and $v_{i+1} v_i \in E_g$ for $i = 0, \ldots, k-1$. Using the $\varepsilon/2$-optimality of f and Lemma 10.6.3, we obtain

$$-\frac{\varepsilon k}{2} \ \leq\ \sum_{i=0}^{k-1} \gamma_p(v_i, v_{i+1}) \ =\ p(v) - p(w) + \sum_{i=0}^{k-1} \gamma(v_i, v_{i+1}).$$

In the same way, the ε-optimality of the original circulation g yields

$$-\varepsilon k \ \leq\ \sum_{i=0}^{k-1} \gamma_q(v_{i+1}, v_i) \ =\ q(w) - q(v) + \sum_{i=0}^{k-1} \gamma(v_{i+1}, v_i).$$

We add the preceding two inequalities and use the antisymmetry of the cost function to obtain

$$-\frac{3\varepsilon k}{2} \ \leq\ p(v) - p(w) + q(w) - q(v).$$

Next we show $p(w) = q(w)$: RELABEL can only be applied for vertices with positive flow excess, so that the original value $q(w)$ of the potential for a vertex w with $e_f(w) < 0$ cannot have changed unless the flow excess has become positive at some point. However, once a vertex has positive flow excess, it can never again acquire negative flow excess because of step (1) in PUSH. Thus $e_f(w) < 0$ indeed implies $p(w) = q(w)$. From this we conclude

$$p(v) \ \geq\ q(v) - \frac{3\varepsilon k}{2} \ \geq\ q(v) - \frac{3\varepsilon |V|}{2}.$$

By Lemma 10.8.3, each RELABEL(v)-operation decreases the original value $q(v)$ of the potential by at least $\varepsilon/2$, so that there cannot be more than $3|V|$ such operations for a given vertex v. □

We can now also treat the saturating PUSH-operations.

Lemma 10.8.7. *Algorithm 10.8.1 involves at most $O(|V||E|)$ saturating* PUSH-*operations.*

Proof. Consider the saturating PUSH-operations for a given edge vw. After such a PUSH(v, w) has been executed, $r_f(v, w) = 0$, so that a further PUSH on vw is possible only if a PUSH(w, v) is executed first. Now the saturating PUSH(v, w) was admissible only if $\gamma_p(v, w) < 0$, whereas a PUSH(w, v) requires the converse condition $\gamma_p(w, v) < 0$ and therefore $\gamma_p(v, w) > 0$. Thus a RELABEL(v)-operation has to occur between any two consecutive saturating PUSH-operations on vw, as this is the only way to decrease

$\gamma_p(v,w) = \gamma(v,w) + p(v) - p(w)$. Now Lemma 10.8.6 shows that at most $O(|V|)$ saturating PUSH-operations may occur on vw during the course of Algorithm 10.8.1. □

As in Section 6.6, the non-saturating PUSH-operations play the crucial role in the complexity of REFINE. We need a lemma to be able to analyze how many non-saturating PUSH-operations occur. Let us call the edges vw of the residual graph G_f which have negative reduced cost $\gamma_p(v,w)$ *admissible edges*, and denote the subdigraph of G_f which contains only the admissible edges – the *admissible graph* – by $G_A = G_A(f)$.

Lemma 10.8.8. *The admissible graph G_A is always acyclic during the course of Algorithm 10.8.1.*

Proof. As mentioned in the proof of Theorem 10.8.4, the pseudoflow f constructed during the initialization (2) to (8) is even 0-optimal, so that the corresponding graph G_A is empty (and hence trivially acyclic). Now a PUSH(v,w) can only be executed if $\gamma_p(v,w) < 0$, so that $\gamma_p(w,v) > 0$. Thus the antiparallel edge wv – which might be added to G_f – is definitely not added to G_A. Hence PUSH-operations do not add edges to G_A, so that G_A stays acyclic. Finally, consider a RELABEL(v)-operation. Before this operation is performed, $\gamma_p(u,v) \geq -\varepsilon/2$ for all $uv \in G_f$. As we saw in Lemma 10.8.3, RELABEL(v) decreases $p(v)$ by at least $\varepsilon/2$, so that $\gamma_p(u,v) \geq 0$ holds after the RELABEL(v). Therefore G_A does not contain any edges with end vertex v, and G_A is still acyclic after the RELABEL(v). □

As in Section 6.6, we could now find an upper bound for the number of non-saturating PUSH-operations. However, we prefer to proceed by performing the admissible operations in a particularly efficient order, and leave the more general result to the reader as an exercise.

Exercise 10.8.9. Show that at most $O(|V|^2|E|)$ non-saturating PUSH-operations occur during Algorithm 10.8.1. Hint: Consider the potential $\Phi = \sum_{v \text{ active}} \Phi(v)$, where $\Phi(v)$ is the number of vertices which are accessible from v in the admissible graph G_A.

We now follow [GoTa90] and present a special version of REFINE called the *first active method*; this is similar to the highest label preflow push algorithm in Section 6.6. Again, we keep adjacency lists A_v and distinguish a *current edge* in each A_v; initially, this is always the first edge of A_v. Moreover, throughout the algorithm, we keep a topological sorting of V with respect to the admissible graph G_A in a list L. As G_A is initially empty, the vertices may be added arbitrarily to L during the initialization phase. Furthermore, we need a *current vertex*: this is always the vertex for which we want to perform the next admissible operation, preferably a PUSH(v,w), but if this is not possible, then a RELABEL(v). Immediately after a RELABEL(v), v is deleted from L and inserted again at the beginning of L. Note that v has

indegree 0 in G_A at this point, as shown in the proof of Lemma 10.8.8. Hence L remains a topological sorting for G_A. In this case, v always stays the current vertex. If v becomes inactive during a PUSH-operation, the next vertex in L is chosen as the current vertex; as L gives a topological sorting of G_A, there can be no active vertex in L before v. These considerations lead to the following algorithm:

Algorithm 10.8.10 (first active method). Let (G, c) be a network with cost function γ as described in Section 10.6, where G is given by adjacency lists A_v. Moreover, let f be an ε-optimal circulation with associated potential p. Finally, let L be a list and *rel* a Boolean variable.

Procedure FAREFINE$(G, c, \gamma, f, \varepsilon, p; f)$

(1) $\varepsilon \leftarrow \varepsilon/2$;
(2) **for** $uv \in E$ **do**
(3) $\quad \gamma_p(u,v) \leftarrow \gamma(u,v) + p(u) - p(v)$;
(4) $\quad$ **if** $\gamma_p(u,v) < 0$
(5) $\quad$ **then** $f(u,v) \leftarrow c(u,v)$, $f(v,u) \leftarrow -c(u,v)$
(6) $\quad\quad r_f(u,v) \leftarrow 0$, $e_f(v,u) \leftarrow c(v,u) - f(v,u)$
(7) $\quad$ **fi**
(8) **od**
(9) **for** $v \in V$ **do** $e(v) \leftarrow \sum_u f(u,v)$ **od**
(10) $L \leftarrow V$;
(11) let v be the first vertex in L;
(12) **while** there exists an active vertex **do**
(13) $\quad$ **if** $e(v) > 0$
(14) $\quad$ **then** rel $\leftarrow$ false; select the first edge in A_v as the current edge;
(15) $\quad\quad$ **repeat**
(16) $\quad\quad$ let vw be the current edge in A_v;
(17) $\quad\quad$ **if** $r_f(v,w) > 0$ **and** $\gamma_p(v,w) < 0$
(18) $\quad\quad$ **then** PUSH$(f, v, w; f)$;
(19) $\quad\quad$ **fi**
(20) $\quad\quad$ **if** $e(v) > 0$ **then**
(21) $\quad\quad\quad$ **if** vw is not the last edge in A_v
(22) $\quad\quad\quad$ **then** choose the next edge in A_v as the current edge
(23) $\quad\quad\quad$ **else** RELABEL$(f, v, p; f, p)$; rel $\leftarrow$ true;
(24) $\quad\quad\quad\quad$ choose the first edge in A_v as the current edge
(25) $\quad\quad\quad$ **fi**
(26) $\quad\quad$ **fi**
(27) $\quad\quad$ **until** $e(v) = 0$ **or** rel = true;
(28) $\quad\quad$ **if** $e(v) = 0$
(29) $\quad\quad$ **then** replace v by the next vertex in L
(30) $\quad\quad$ **else** move v to the beginning of L
(31) $\quad\quad$ **fi**
(32) $\quad$ **else** replace v by the next vertex in L

(33) **fi**
(34) **od**

Theorem 10.8.11. *Algorithm 10.8.10 constructs with complexity* $O(|V|^3)$ *an* $\varepsilon/2$*-optimal circulation* f *on* (G, c).

Proof. As Algorithm 10.8.10 is a special version of Algorithm 10.8.1, Theorem 10.8.4 implies that it is correct, provided that it terminates. By Lemma 10.8.6, there are at most $O(|V|^2)$ RELABEL-operations during the execution of the algorithm; each of them needs at most $O(|V|)$ steps. Moreover, by Lemma 10.8.7, there are at most $O(|V||E|)$ saturating PUSH-operations, each of which takes only $O(1)$ steps. Thus it suffices to show that there are altogether at most $O(|V|^3)$ non-saturating PUSH-operations.

As noted before, the list L contains a topological sorting of the vertices with respect to the admissible graph G_A throughout the algorithm. By the word *phase* we refer to the the sequence of operations between two consecutive RELABEL-operations, or between the beginning of the algorithm and the first RELABEL-operation, or after the last RELABEL and the termination of the algorithm. By Lemma 10.8.6, there are at most $O(|V|^2)$ phases. At the beginning of each phase, v is always the first vertex of L – initially because of (11), later because of (30). Of course, the algorithm may examine at most all $|V|$ vertices in L before either the next RELABEL-operation is performed or the algorithm terminates. For each vertex v, there can be at most one non-saturating PUSH-operation during a given phase: after such a PUSH, $e(v) = 0$ so that v is replaced by the next vertex in L. This yields the desired bound of at most $O(|V|)$ non-saturating PUSH-operations during each phase. □

Using the procedure FAREFINE above in Algorithm 10.7.1 instead of REFINE, we get an algorithm which constructs an optimal circulation for a given network with the complexity stated in Theorem 10.7.8.

10.9 The minimum mean cycle cancelling algorithm

In this section, we shall return to the algorithm of Klein and show that an appropriate specification yields a polynomial algorithm, a result due to Goldberg and Tarjan [GoTa89]. The complexity one obtains is inferior to the complexity achieved in Theorem 10.7.7; however, the modified algorithm of Klein is particularly simple and intuitive.

Let us first consider a specialization of the algorithm of Klein to flow networks as in Chapter 6. As in Example 10.1.1, we add the return arc $r = ts$ to the flow network $N = (G, c, s, t)$; put $\gamma(r) = -1$ and $\gamma(e) = 0$ for all other edges e; and consider the corresponding problem of finding an optimal circulation. Then a flow f of value w on N corresponds to a circulation f' on $G' = G \cup \{r\}$ with cost $-w(f)$. Now let (H, w) be the auxiliary network

with respect to f', as constructed in Algorithm 10.4.1. Obviously, the only cycles of negative length are cycles containing the return arc, and these cycles correspond precisely to the augmenting paths in G with respect to f – that is, to paths from s to t in the auxiliary network $N'(f)$; see Section 6.3. It is now easily seen that the algorithm of Klein reduces to the labelling algorithm of Ford and Fulkerson (Algorithm 6.1.7) for determining a maximal flow. As shown in Section 6.1, the algorithm of Klein is therefore not polynomial even if all input data are integral.

As we have seen in Section 6.2, the algorithm of Ford and Fulkerson becomes polynomial if the augmenting paths are chosen in a clever way: always select an augmenting path P of shortest length in $N'(f)$. This suggests interpreting this strategy in terms of the associated circulations, and then trying to generalize it to arbitrary circulations. As already mentioned, P corresponds to a cycle C of negative length in (H, w). Note that all these cycles have the same length, namely $w(C) = -1$, which might be disappointing. Fortunately, the length $|P|$ of P is reflected in the mean weight $m(C)$ of C:

$$m(C) \;=\; \frac{w(C)}{|C|} \;=\; -\frac{1}{|P|+1}.$$

Thus an augmenting path of shortest length in $N'(f)$ corresponds to a cycle with minimum cycle mean $\mu(H, w)$. This motivates the strategy suggested by Goldberg and Tarjan: always cancel a (negative) cycle of minimum cycle mean in order to improve the present circulation f; recall that such a cycle can be determined efficiently by the method of Karp described in Section 10.6. We will see that the resulting algorithm is indeed polynomial – but this will require considerable effort.

Algorithm 10.9.1 (minimum mean cycle canceling algorithm). Let G be a digraph with capacity constraints b and c and a cost function γ. The algorithm decides whether an admissible circulation exists; if this is the case, it constructs an optimal circulation.

Procedure MMCC$(G, b, c, \gamma;$ legal, $f)$

```
(1)  LEGCIRC(G, b, c, γ; legal, f);
(2)  if legal = true then repeat
(3)      E' ← ∅;
(4)      for e = uv ∈ E do
(5)          if f(e) < c(e)
(6)          then E' ← E' ∪ {e}; tp(e) ← 1; c'(e) ← c(e) − f(e);
                 w(e) ← γ(e) fi
(7)          if b(e) < f(e)
(8)          then e' ← vu; E' ← E' ∪ {e'}; tp(e') ← 2;
                 c'(e') ← f(e) − b(e); w(e') ← −γ(e)
(9)          fi
(10)     od
```

(11) $H \leftarrow (V, E')$;
(12) MEANCYCLE $(H, w; \mu, C, \text{acyclic})$
(13) **if** acyclic = false **and** $\mu < 0$
(14) **then** $\delta \leftarrow \min\{c'(e) : e \in C\}$;
(15) **for** $e \in C$ **do**
(16) **if** $tp(e) = 1$ **then** $f(e) \leftarrow f(e) + \delta$ **else** $f(e) \leftarrow f(e) - \delta$ **fi**
(17) **od**
(18) **fi**
(19) **until** acyclic = true **or** $\mu \geq 0$
(20) **fi**

Here the procedure MEANCYCLE is the algorithm described in Exercise 10.6.13. As usual, we refer to a change of f along a cycle C as in steps (14) to (17) above as *cancelling the cycle* C.

The rest of this section is devoted to showing that Algorithm 10.9.1 is indeed polynomial. We now think of the original network (G_0, b, c) and the corresponding circulations as transformed into the form (G, c) described in Construction 10.6.1 and Algorithm 10.7.1, so that we may apply the results of Sections 10.6 and 10.7. Even though the MMCC-algorithm does *not* use the technique of successive approximation, we nevertheless need the theory of ε-optimality for analyzing it.

We saw in Section 10.6 that each circulation f is ε-tight for some value $\varepsilon \geq 0$; let us denote this number by $\varepsilon(f)$. The following lemma shows that cancelling a cycle of minimum cycle mean does not increase this parameter.

Lemma 10.9.2. *Let f be an ε-tight circulation on (G, c) with respect to the cost function γ, where $\varepsilon > 0$. Moreover, let C be a directed cycle of minimum cycle mean in the residual graph G_f. Then the circulation g obtained by cancelling C satisfies $\varepsilon(g) \leq \varepsilon(f) = \varepsilon$.*

Proof. Let p be a potential corresponding to f. Then, by Lemma 10.7.5,

$$\gamma_p(u, v) = -\varepsilon \qquad \text{for all } uv \in C. \tag{10.16}$$

We obtain the residual graph G_g from G_f by deleting some edges of C and adding some edges which are antiparallel to edges of C. Now (10.16) implies

$$\gamma_p(v, u) = -\gamma_p(u, v) = \varepsilon > 0$$

for edges $uv \in C$, so that the condition $\gamma_p(u, v) \geq -\varepsilon$ also holds for all edges uv in G_g. Hence g is ε-optimal with respect to the potential p, and therefore $\varepsilon(g) \leq \varepsilon = \varepsilon(f)$. □

Now it is possible that cancelling C does not lead to an improvement of the tightness: $\varepsilon(g) = \varepsilon(f)$ in Lemma 10.9.2 may occur. However, the next lemma shows that this cannot happen too often.

Lemma 10.9.3. *Let f be a circulation on (G, c) which is ε-tight with respect to the cost function γ. Suppose g is a circulation obtained from f by cancelling $|E|$ cycles of minimum cycle mean. Then*

$$\varepsilon(g) \;\leq\; (1 - \frac{1}{|V|})\varepsilon.$$

Proof. Let p be a potential corresponding to f:

$$\gamma_p(u, v) \;\geq\; -\varepsilon \qquad \text{for all } uv \in E_f.$$

As we saw in the proof of Lemma 10.9.2, all edges added to G_f when cancelling a cycle of minimum cycle mean have positive reduced cost $\gamma_p(u, v)$. On the other hand, at least one edge e (for which the minimum in step (14) of Algorithm 10.9.1 is achieved) is deleted from G_f. Also note that p always remains unchanged. Now we distinguish two cases.

Case 1: All $|E|$ cycles which were cancelled to obtain g consist of edges e with $\gamma_p(e) < 0$ only. Then all edges added to G_f by these cancellations have positive reduced cost. As at least one edge with negative reduced cost is deleted for each cancellation, only edges with nonnegative reduced cost can remain in the residual graph after these $|E|$ cancellations. Therefore, g is optimal: $\varepsilon(g) = 0$, and the assertion holds.

Case 2: At least one of the cycles cancelled contains some edge with nonnegative reduced cost with respect to p. Let C be the first cancelled cycle with this property. All edges e added to G_f before C was cancelled have positive reduced cost $\gamma_p(e)$. Hence we have

$$\gamma_p(e) \geq -\varepsilon \;\text{ for all } e \in C \quad \text{and} \quad \gamma_p(e_0) \geq 0 \;\text{ for some edge } e_0 \in C.$$

Therefore

$$\begin{aligned} m(C) \;&=\; \frac{1}{|C|} \sum_{e \in C} \gamma(e) \;=\; \frac{1}{|C|} \sum_{e \in C} \gamma_p(e) \\ &\geq\; \frac{-(|C| - 1)\varepsilon}{|C|} \;\geq\; -(1 - \frac{1}{|V|})\varepsilon. \end{aligned}$$

Let h denote the circulation which has been changed by cancelling C. Then

$$\mu(G_h, \gamma) \;=\; m(C) \;\geq\; -(1 - \frac{1}{|V|})\varepsilon,$$

and hence, by Theorem 10.6.10,

$$\varepsilon(h) \;=\; -\mu(G_h, \gamma) \;\leq\; (1 - \frac{1}{|V|})\varepsilon.$$

Repeated application of Lemma 10.9.2 yields $\varepsilon(g) \leq \varepsilon(h)$, which implies the assertion. □

We need one further simple lemma.

Lemma 10.9.4. *Let m be a positive integer, and let $(y_k)_{k\in\mathbb{N}}$ be a sequence of nonnegative reals satisfying the condition*

$$y_{k+1} \;\leq\; (1-\frac{1}{m})y_k \quad \textit{for all } k \in \mathbb{N}.$$

Then $y_{k+m} \leq y_k/2$ for all $k \in \mathbb{N}$.

Proof. By hypothesis,

$$y_k \;\geq\; y_{k+1} + \frac{y_{k+1}}{m-1} \quad \text{for all } k,$$

so that

$$\begin{aligned} y_k \;&\geq\; y_{k+1} + \frac{y_{k+1}}{m-1} \\ &\geq\; (y_{k+2} + \frac{y_{k+2}}{m-1}) + \frac{y_{k+1}}{m-1} \;\geq\; y_{k+2} + \frac{2y_{k+2}}{m-1} \\ &\geq\; \ldots \quad \geq y_{k+m} + \frac{my_{k+m}}{m-1} \;\geq\; 2y_{k+m}. \end{aligned}$$

□

Theorem 10.9.5. *Algorithm 10.9.1 determines in $O(|V||E|^2 \log|V|)$ iterations an optimal circulation on (G, b, c).*

Proof. Put $k = |V||E|\lceil\log|V|+1\rceil$ and divide the iterations of Algorithm 10.9.1 into phases of k subsequent cancellations. We claim that at least one further edge of G becomes ε-fixed (for some appropriate ε) during each phase. This yields the assertion, because the algorithm has to terminate at the latest after all edges have become ε-fixed.

Now let f_0 and f_k be the circulations constructed directly before the first cancellation and directly after the last cancellation of some phase, respectively. Put $\varepsilon = \varepsilon(f_0)$ and $\varepsilon' = \varepsilon(f_k)$, and let p be a potential corresponding to f_k:

$$\gamma_p(v,w) \;\geq\; -\varepsilon' \quad \text{for all } vw \in G_{f_k}.$$

By Lemma 10.9.3, any $|E|$ subsequent cancellations decrease $\varepsilon(f)$ by at least a factor of $1 - 1/|V|$. Using Lemma 10.9.4, this implies that any $|V||E|$ subsequent cancellations decrease $\varepsilon(f)$ by at least a factor of $1/2$. Therefore,

$$\varepsilon' \;\leq\; \varepsilon \times (\frac{1}{2})^{\lceil\log|V|+1\rceil} \;\leq\; \frac{\varepsilon}{2|V|},$$

so that

$$-\varepsilon \;\leq\; -2|V|\varepsilon'. \tag{10.17}$$

Now let C be the cycle which is used first during the phase under consideration: f_0 is changed cancelling C. Then, by Theorem 10.6.10,

$$m(C) \;=\; -\varepsilon \quad \text{in } (G_{f_0}, \gamma).$$

By Lemma 10.6.3, also $m(C) = -\varepsilon$ in (G_{f_0}, γ_p), and hence C contains an edge e with $\gamma_p(e) \leq -\varepsilon$. Then (10.17) yields $\gamma_p(e) \leq -2|V|\varepsilon'$, and e is ε'-fixed by Corollary 10.7.4. On the other hand, e was not ε-fixed, as e is contained in the cycle C which was cancelled when f_0 was changed. Thus at least one further edge becomes δ-fixed (for some appropriate value of δ) during each phase. □

Exercise 10.9.6. Assume that the cost function γ is integral. Show that Algorithm 10.9.1 terminates after $O(|V||E| \log(|V|C))$ iterations, where

$$C \;=\; \max\{|\gamma(u,v)| : uv \in E\}\,.$$

Using Exercise 10.6.13, Theorem 10.9.5, and Exercise 10.9.6 yields the following result.

Theorem 10.9.7. *Algorithm 10.9.1 determines in $O(|V|^2|E|^3 \log|V|)$ steps an optimal circulation on (G, b, c). If γ is integral, the complexity is also bounded by $O(|V|^2|E|^2 \log(|V|C))$.* □

The reader may find a more detailed examination of the number of cancellations needed by Algorithm 10.9.1 in [RaGo91]. Using appropriate data structures and making some modifications in the way the negative cycles are chosen, the bounds of Theorem 10.9.7 can be improved: one may obtain a complexity of $O(|V||E|^2(\log|V|)^2)$; see [GoTa89]. There also exist polynomial algorithms which work with cancellations of *cuts*: these algorithms are – in the sense of linear programming – dual to the algorithms where cycles are cancelled; see [ErMcC93].

10.10 Some further problems

In this section, we discuss some further problems which can be dealt with using optimal circulations or optimal flows. We will also mention some generalizations of the problems treated so far; however, we have to refer to the literature for more information on most of these problems. An even more general version of the following problem will be studied in detail in Chapter 11.

Example 10.10.1 (transshipment problem). Let $G = (V, E)$ be a digraph with a nonnegative capacity function $c : E \to \mathbb{R}$ and a nonnegative cost function $\gamma : E \to \mathbb{R}$. Moreover, let X and Y be disjoint subsets of V; we call the elements of X *sources* and the elements of Y *sinks*, as in Section 7.7. Again, we associate with each source x a *supply* $a(x)$ and with each sink y a *demand* $b(y)$, where the functions a and b are nonnegative. As in the supply and demand problem of Section 7.7, we require a *feasible flow*[13] on

[13] Sometimes, there are upper bounds placed also on the capacities of the edges.

(G, c): a mapping $f: E \to \mathbb{R}$ satisfying conditions (ZF 1) to (ZF 4) of Section 7.7. Moreover, we want to find an *optimal flow* among all feasible flows; that is, a flow of minimal cost with respect to γ. This *transshipment problem* is the weighted version of the supply and demand problem. Again, we add a new source s, a new sink t, all edges sx with capacity $c(sx) = a(x)$, and all edges yt with capacity $c(yt) = b(y)$. We also extend the cost function γ by putting $\gamma(sx) = 0$ and $\gamma(yt) = 0$. Then an optimal flow of value $\sum b(y)$ on the resulting flow network N gives a solution for our problem. To find such a solution, we may, for example, use the algorithm of Busacker and Gowen presented in Section 10.5.

Example 10.10.2 (transportation problem). A transshipment problem for which $V = X \dot\cup Y$ holds is called a *transportation problem*. In this case, there are no *intermediate nodes*: each vertex of V is either a source or a sink. If G is the complete bipartite graph on $X \dot\cup Y$, the problem is called a *Hitchcock problem*; see [Hit41]. Note that the assignment problem of Example 10.1.4 is a special Hitchcock problem: it is the case with $|X| = |Y|$ where all capacities and all the values $a(x)$ and $b(y)$ are equal to 1.

We have seen that the Hitchcock problem is a very special case of the problem of finding optimal flows on a flow network. Conversely, it can be shown that the general problem of finding optimal flows can be transformed to a Hitchcock problem (even without capacity constraints) on an appropriate bipartite graph; see, for example, [Law76, §4.14].

The transshipment problem (with or without capacity constraints) is often solved in practice using a special version of the simplex algorithm of linear programming, namely the so-called *network simplex algorithm* which we will study in the next chapter. A very good presentation of this method can also be found in part III of [Chv83], a book that is recommendable in general.[14] Although the network simplex method can be rather bad when applied to certain pathological networks [Zad73a], it is spectacularly successful in practice. As Chvátal puts it: 'It takes just a few minutes to solve a typical problem with thousands of nodes and tens of thousands of arcs; even problems ten times as large are solved routinely.' Meanwhile, polynomial variants of the network simplex method have been found; see [OrPT93], [Orl97], and [Tar97].

Finally, we mention some generalizations of the flow problems treated in this book. In Section 12.2, we consider *multiterminal problems*, where we want to determine the maximal flow values between all pairs of vertices; usually, the graph underlying such a network is assumed to be undirected.

More about the following three generalizations can be found in [FoFu62], [GoMi84], and [AhMO93]. For some practical problems, it makes sense to

[14] The author of the present book thinks that the most intuitive way to become acquainted with problems of combinatorial optimization is the presentation in a graph theory context; however, the theory of linear programming is indispensable for further study.

consider *flows with gains or losses*: the quantity of flow entering an edge at vertex u is changed by a factor m_u while passing through that edge. This may serve as a model for exchanging currencies[15], or for losses in a water supply system due to evaporation. A weakly polynomial algorithm for this problem can be found in [GoPT91].

One also considers networks on which different flows occur simultaneously without intermingling (*multicommodity flows*); see, for example, [Lom85]. A polynomial algorithm for this problem was given by Tardos [Tar86].

Finally, one also studies *dynamic flows*: here transversal times are assigned to the edges; this is definitely relevant for traffic networks. This problem can be reduced to flows in the usual sense; see [FoFu58a, FoFu62].

We also mention an interesting collection of papers concerning various network problems: [KlPh86].

A detailed discussion how actual problems from the practice of operations research may be modelled as network problems is beyond the scope of this book; at least, we have seen a few examples already. Modelling is an extremely important – and by no means trivial – task, and it has to be accomplished before any of the mathematical algorithms presented in this book can be applied. We recommend the monograph [GlKP92] for more about this subject, and the references given there for further actual case studies. An interesting more recent application is in [JaKR93]: two models for deciding between delays and cancellations of flights when planes cannot be used as scheduled. In particular, an optimal flow problem is solved in this context using the algorithm of Busacker and Gowen.

10.11 An application: Graphical codes

We close this chapter with an application of the material covered in Section 10.3 to coding theory: the cycle space of a graph gives rise to an (often interesting) binary code. Our presentation is based on the author's tutorial paper [JuVa96].

We begin with a brief introduction to codes; the interested reader may consult one of the standard text books for more details; see, for instance, [MacSl77] or [vLi99]. In the most general sense, a *code* of *length* n is just a subset of some set of the form S^n, where S is any finite set, the underlying *alphabet*. So we once again consider words of length n over S, which are called *codewords* in this context. In most applications, one uses *binary* codes, that is, the alphabet is $S = \{0, 1\}$; we shall mostly restrict ourselves to this case.[16]

[15] [Gro85, §8.2] shows an actual example where a chain of transactions resulting in a gain occurs.

[16] In everyday language, a *code* usually means a way of secret communication: for instance, one speaks of *breaking a code*. However, the corresponding part of mathematics is *cryptography*, whereas *coding theory* deals with the problem of ensuring

The idea of coding is now quite simple: one first transforms the data to be transmitted (or stored) into binary strings of a fixed length k. Let us explain this via an example. Suppose we want to transmit (or store) an image digitally. Then we would discretize the image by dividing it into small squares called *pixels* and – in the simple case of black and white pictures – associate with each pixel a value between, say, 0 and 63 measuring the darkness of the pixel: 0 would stand for white, 63 for black. Each such value then corresponds – in binary representation – to a 6-tuple over $S = \{0, 1\}$, and we can represent the entire image as a sequence of such 6-tuples.

However, just transmitting or storing these 6-tuples would not work. For example, when transmitting an image from a satellite to earth, the weak signal received would certainly contain quite a few errors, and this would be fatal. Just one error could, for instance, transform the 6-tuple (000 000) representing white into (100 000) representing a medium dark gray! Of course, similar problems arise in digital media such as compact discs, where dust and scratches lead to incorrect readouts.

To cope with these problems, one needs to introduce redundancy. For this purpose one chooses an injection α from S^6 into S^{32}, to give a concrete example.[17] The images of 6-tuples under α are the codewords. If one receives a signal which *is* a codeword, it is decoded into its pre-image. If the received word $\mathbf{w}$ is not a codeword, one chooses a codeword which differs from $\mathbf{w}$ in the smallest possible number of entries and decodes $\mathbf{w}$ into its pre-image, as it is more likely that a bit is transmitted correctly than that it is changed. Clearly, it is now important to select the injection α in a clever way: any two codewords should differ from one another in a large number of positions. Formally, we arrive at the following definition.

Definition 10.11.1. Let $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_n)$ be any two words of length n over the alphabet S. Then the function d defined by

---

that information can be recovered accurately after transmission or storage, in spite of possible loss or corruption of data; there is no aspect of secrecy involved. To use the words of Claude Shannon [Sha48], the founder of *information theory* (who, by the way, also put cryptography onto a sound mathematical basis in his paper [Sha49b]), of which coding theory is one part:

> The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point.

Codes in this sense constitute a spectacularly successful part of applied mathematics: satellite images, CD's, DVD's, digital TV, mobile phones all need codes (to mention just a few examples). In this context, we also recommend another introduction to coding theory, namely [VaOo89], where the reader may find a nice description of the mathematical principles underlying compact discs.

[17] This is precisely the approach used for transmitting pictures from one of the early satellites, namely *Mariner 9*. The injection chosen in this case made it possible to correct up to seven errors per codeword, which was sufficient to make an incorrect decoding very unlikely; see [Pos69] for details.

$$d(\mathbf{x}, \mathbf{y}) \;=\; |\{j = 1, \ldots, n\colon\; x_j \neq y_j\}|$$

is called the *Hamming distance* on S^n. In case $\mathbf{x} \in C \subseteq S^n$ and $d(\mathbf{x}, \mathbf{y}) = e$, we say that $\mathbf{y}$ arises from the codeword $\mathbf{x}$ by e *errors* and refer to the set $E = \{j = 1, \ldots, n\colon\; x_j \neq y_j\}$ as the *error pattern*.[18] Moreover, one calls the distance $d(\mathbf{x}, \mathbf{0})$ of a word $\mathbf{x}$ from the zero vector $\mathbf{0}$ the *weight* $w(\mathbf{x})$ of $\mathbf{x}$.

Exercise 10.11.2. Verify that the Hamming distance indeed satisfies the axioms in Section 3.2, so that (S^n, d) is a finite metric space.

It is usual to define the following *parameters* of a code C: the *length* n, the cardinality M, the *minimum distance* d – that is, the minimal value $d(\mathbf{x}, \mathbf{y})$, taken over all pairs of distinct codewords – and the size q of the underlying alphabet S. One then speaks of an (n, M, d, q) code. The importance of the parameter d is due to the following simple but fundamental result:

Lemma 10.11.3. *Let C be a code with minimum distance d. Assume that some codeword $\mathbf{x}$ is transmitted, and that the word $\mathbf{y}$ received arises from $\mathbf{x}$ by e errors, where $2e + 1 \leq d$. Then $\mathbf{x}$ can be recovered from $\mathbf{y}$: it is the unique codeword having minimal distance from $\mathbf{y}$.*

Proof. Assume otherwise. Then there exists a codeword $\mathbf{z} \neq \mathbf{x}$ satisfying $d(\mathbf{z}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{y})$. But then, by Exercise 10.11.2,

$$d(\mathbf{x}, \mathbf{z}) \;\leq\; d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z}) \;\leq\; 2d(\mathbf{x}, \mathbf{y}) \;=\; 2e \;<\; d,$$

a contradiction. □

In view of Lemma 10.11.3, a code with minimum distance $d = 2t + 1$ or $d = 2t + 2$ is also called a *t-error correcting code*. It is now clear that one would like, for fixed values of n and q, say, to make both d and M large: we want to be able to correct many errors, but we also want to be able to encode many messages. Of course, these two goals are contradictory. Hence it is of interest to study the function $A(n, d, q)$ defined as the largest possible size M of an (n, M, d, q) code; the determination of this function is one of the major problems in coding theory. We will not deal with this problem here in general, but restrict attention to a special class of codes which is of particular practical importance.

A code is called *linear* if it is a linear subspace of a vector space $V = F^n$, where F is a finite field.[19] In particular, a binary linear code is a subspace of

[18] Note that we may reconstruct $\mathbf{x}$ uniquely from $\mathbf{y}$ and the error pattern E, provided that we are in the binary case.

[19] Finite fields are an important tool in many applications, not just in coding theory, but also in cryptography, signal processing, geometry, and design theory, as well as in algebraic graph theory and matroid theory, to mention just some examples. It is known that the cardinality of a finite field has to be a prime power q, and that there is – up to isomorphism – exactly one finite field with q elements (for

a vector space over $\mathbb{Z}_2$. If a code C of length n over $GF(q)$ is a k-dimensional subspace, it will consist of $M = q^k$ codewords; it is customary to denote such a code as an $[n, k, d; q]$ code. In the binary case, it is usual to omit the parameter q and simply speak of an $[n, k, d]$ code. Note that we may now simplify the determination of the minimum distance d by just computing the *minimum weight* instead, that is, the minimal value $w(\mathbf{x})$, taken over all codewords; this follows from the obvious fact $d(\mathbf{x}, \mathbf{y}) = w(\mathbf{x} - \mathbf{y})$.

As for arbitrary codes, we would like to find linear codes which, for fixed n and q, have large k and d. Again, we cannot deal with this problem in any systematic way; for instance, we will entirely ignore the fundamental problem of finding upper bounds for k when we also fix d. What we will do is using the cycle spaces of graphs to construct some interesting (and, often, actually very good) binary linear codes.

Thus let $G = (V, E)$ be a graph with n vertices and m edges. An *even subgraph* of G (sometimes simply called an *even graph* if there is no danger of confusion) is a spanning subgraph of G in which each vertex has even degree (where degree 0 is allowed). Similarly, an *odd subgraph* of G is a spanning subgraph in which each vertex has odd degree. It is easy to check that the set of all even subgraphs of G is closed under the symmetric difference of subgraphs, where spanning subgraphs are simply considered as subsets of E. We will denote this set by $C_E(G)$ and consider it as a binary linear code.

To this purpose, we use an approach similar to that of Section 10.3 and view the set of all spanning subgraphs of G as the binary vector space $V(m, 2)$ of all m-tuples with entries from $\mathbb{Z}_2$: we consider the coordinate positions to be indexed by the edges of G (in some fixed ordering); then each spanning subgraph is associated with the corresponding binary characteristic vector of length m which has an entry 1 in position e if and only if e belongs to the given subgraph (and 0 otherwise). Note that $C_E(G)$ then corresponds to a subspace of $V(m, 2)$, as the symmetric difference of spanning subgraphs corresponds to the mod 2 addition of their incidence vectors; by abuse of notation, we will denote this subspace again by $C_E(G)$. Thus $C_E(G)$ can indeed be viewed as a binary linear code, an approach going back to work of Bredeson and Hakimi [BrHa67, HaBr68].

One calls $C_E(G)$ the *even graphical code* associated with the graph G; for the sake of simplicity, we will usually assume that G is connected. As it turns out, it is not too difficult to determine the parameters of a graphical code. We first observe that $C_E(G)$ is just the binary analogue of the cycle space of G considered in Section 10.3, as any even subgraph H of G is the disjoint union of Eulerian subgraphs on its connected components, by Theorem 1.3.1. The

---

every prime power q), which is usually denoted by $GF(q)$. For background on finite fields, we refer the reader to the standard text book [LiNi94] or to [McEl87] for an introduction requiring very little algebraic knowledge; both of these books also discuss some interesting applications.

reader should have no problem establishing the following analogue of Theorem 10.3.6:

Exercise 10.11.4. Let G be a digraph with n vertices, m edges, and p connected components. Show that the vector space $C_E(G)$ has dimension $\nu(G) = m - n + p$.

Theorem 10.11.5. *Let G be a connected graph with m edges on n vertices, and let g be the girth of G. Then $C_E(G)$ is a binary $[m, m-n+1, g]$-code.*

Proof. In view of Exercise 10.11.4, it only remains to determine the minimum distance d of $C_E(G)$. It is clear that the minimum weight of a vector in $C_E(G)$ is the smallest cardinality of a cycle in G, that is the girth g of G. (Recall that we have already seen this important graph theoretic parameter in Section 1.5). □

We remark that the dual of $C_E(G)$ – that is, the orthogonal complement in $V(m, 2)$ with respect to the standard inner product – is the binary analogue of the bond space of G studied in Exercise 10.3.8. Thus G also gives us a second type of binary code; however, in general, such codes seems to be of inferior quality, see [HaFr65].

Let us give a few examples illustrating the construction of binary codes according to Theorem 10.11.5:

Example 10.11.6. The complete graph K_n gives rise to a binary linear code with parameters $[n(n-1)/2, (n-1)(n-2)/2, 3]$: we have $m = n(n-1)/2$ edges, n vertices, and the girth is 3.

We shall return to these examples from time to time. However, our main example will be the Petersen graph (see Figure 1.12) and its code; this example will be studied in more detail.

Example 10.11.7. Let G be the Petersen graph. Then $P = C_E(G)$ is a binary linear [15,6,5] code: G has $m = 15$ edges, $n = 10$ vertices, and the girth is 5.

The *Petersen code* is in fact one of the standard examples of graphical codes; it is also used in the book of van Lint and Wilson [vLiWi01]. In coding theory, one is often interested in the *weight enumerator* of a code, which is defined as the polynomial

$$A_C(x) \;=\; A_0 + A_1 x + A_2 x^2 + \ldots + A_n x^n,$$

where A_i is the number of codewords of weight i. We shall now determine the weight enumerator of the Petersen code P.

Example 10.11.8. The Petersen code has weight enumerator

$$A_P(x) \;=\; 1 + 12x^5 + 10x^6 + 15x^8 + 20x^9 + 6x^{10}. \tag{10.18}$$

We will establish this assertion by describing all codewords of P. To this end, it is helpful to recall that the automorphism group of the Petersen graph G is both vertex- and edge-transitive; see Exercise 1.5.12 and its solution. We shall also make use of the representation of the Petersen graph given in Figure B.4; in particular, we recall that two vertices are adjacent if and only if their labelling 2-subsets of $\{1, \ldots, 5\}$ are disjoint. We can now describe the words in P.

The codewords of weight 5 are necessarily 5-cycles. One easily checks that a fixed edge is in exactly four such cycles; hence we have altogether $4 \times 15/5 = 12$ codewords of weight 5. Two cycles through the edge $\{15, 23\}$ are indicated in Figure 10.5; a third one is obtained from the cycle having dashed edges by symmetry, and the final one is the outer cycle. A similar argument shows that there are exactly four 6-cycles through a fixed edge and hence altogether ten 6-cycles.

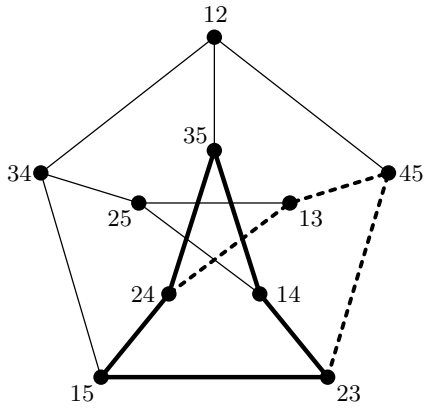


Fig. 10.5. 5-cycles in the Petersen graph

A codeword of weight eight has to be an 8-cycle, since there are no 4-cycles; these can be constructed as follows. Given an edge e of G, select the two edges disjoint to e for each of the four vertices adjacent to one of the end vertices of e; see the heavy edges in Figure 10.6, where e is the dashed edge. These eight edges form an 8-cycle. As the automorphism group of G is edge-transitive, we obtain exactly fifteen 8-cycles in this way.

Similarly, a codeword of weight nine has to be a 9-cycle; these can be described as follows. Given a vertex v of G, select the two edges not through v for each of the three neighbors of v; see the heavy edges in Figure 10.7, where v is the fat vertex. These six edges can then be joined to a 9-cycle in exactly two ways which are indicated by the dashed edges and the gray edges, respectively. As the automorphism group of G is vertex-transitive, we obtain exactly twenty 9-cycles in this way.

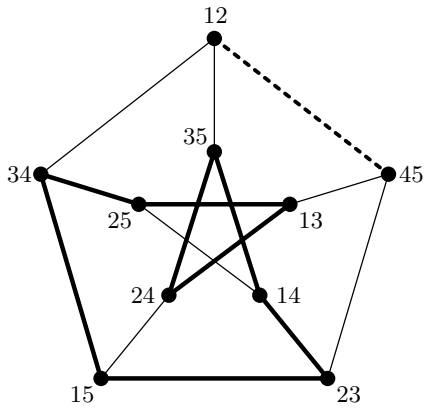


Fig. 10.6. 8-cycles in the Petersen graph

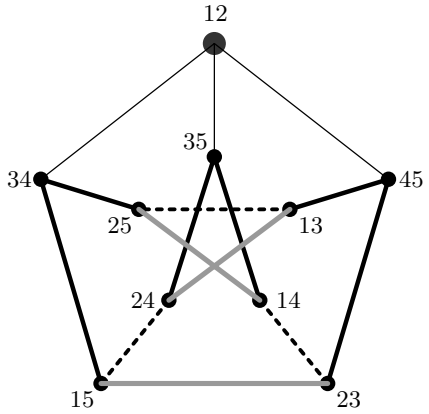


Fig. 10.7. 9-cycles in the Petersen graph

Finally, a codeword of weight 10 has to be the union of two disjoint 5-cycles; each edge is in exactly four such codewords, giving altogether six words of weight 10. For the edge $\{15, 23\}$, the most obvious codeword of this type is the union of the outer cycle with the inner pentagram. A less obvious codeword of weight 10 through this edge is indicated in Figure 10.8, and the remaining two codewords of weight 10 through the edge $\{15, 23\}$ are obtained via rotations.

Thus we have obtained 63 codewords in P; the final one is, of course, the empty subgraph.

Next, we shall discuss a problem first investigated in [HaBr68], namely the possibility of augmenting an even graphical code to a code of larger dimension

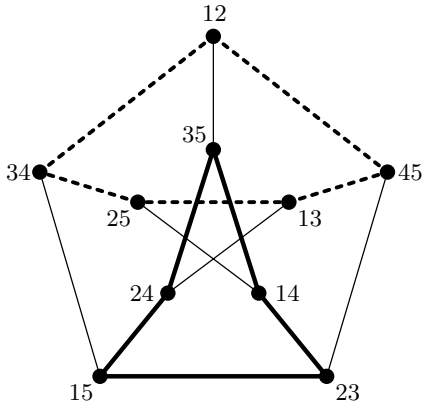


Fig. 10.8. Disjoint 5-cycles in the Petersen graph

while keeping the minimum distance unchanged. More precisely, we shall be interested in augmenting a code $C_E(G)$ by adjoining further subgraphs of G which will give a particularly simple and intuitive way of achieving the augmentation; any such code will be called a *graphical code* based on G. This is the approach proposed in [JuVa97].

Since any subgraph not yet in $C_E(G)$ necessarily contains vertices of odd degree, we first require information about the *odd degree patterns* of subgraphs of G. We may view these as binary vectors of length n as follows: we label the coordinate positions of $V(n,2)$ with the vertices of G (in an arbitrary but fixed way) and put an entry 1 in position v if and only if v has odd degree in the given subgraph. The following auxiliary result is simple but fundamental for our problem.

Lemma 10.11.9. *Let G be a connected graph on n vertices. Then the odd degree patterns of subgraphs of G form the subspace $V_e(n,2)$ of $V(n,2)$ consisting of all even weight vectors in $V(n,2)$.*

Proof. By Lemma 1.1.1, the number of vertices of odd degree in any given graph is even; hence only even weight vectors can occur. Next, note that the mod 2 sum of the odd degree patterns associated with two subgraphs H and H' of G is just the odd degree pattern associated with the symmetric difference of these subgraphs. Hence the odd degree patterns form a subspace of $V(n,2)$.

It remains to show that any subset W of even cardinality $2k$ of the vertex set V can indeed serve as an odd degree pattern. To see this, split W into k pairs, say $\{x_i, y_i\}$ with $i = 1, \ldots, k$, and select an arbitrary path P_i with end vertices x_i and y_i for $i = 1, \ldots, k$. Then the symmetric difference of these k paths has odd degree pattern W. □

Of course, it is also of interest to exhibit an efficient method for actually constructing a spanning subgraph with a prescribed odd degree pattern. This

can be easily done by first conducting a breadth first search on G, say with start vertex s. As we saw in Section 3.3, this yields (in time $O(m)$) a spanning tree for G such that, for any given vertex v, the unique path from v to s in T is actually a shortest path from v to s in G. Now let W be any desired odd degree pattern of cardinality $2k$; then we may just take the symmetric difference of the $2k$ paths in T joining the vertices in W to s to obtain a subgraph S with the correct odd degree pattern.

After these preparations we can solve the problem of augmenting a given even graphical code $C = C_E(G)$ with parameters $[m, m - n + 1, g]$ based on a graph G about which we do not assume to have any further structural information beyond the parameters n, m, and g. Let S be any subgraph of G. We want to check if we may adjoin S to C without causing the minimum distance of the code C^* generated by C and S to decrease.

Let W be the odd degree pattern of S, and let w be the weight of W. We require a bound on the weight $w(H + S)$ of an arbitrary subgraph $H + S$ with $H \in C$ (which, of course, likewise has odd degree pattern W). Note that $H + S$ might just consist of $w/2$ independent edges – that is, $H + S$ might be a matching of size $w/2$ in G – since we assume to have no extra knowledge about G; this obviously gives the smallest conceivable weight w. Hence, if we want to be certain that C^* still has minimum weight g, we have to assume the inequality $w \geq 2g$. Clearly, this necessary condition on the choice of S is also sufficient, proving the following result due to [JuVa97].

Lemma 10.11.10. *Consider an even graphical code $C = C_E(G)$ with parameters $[m, m - n + 1, g]$ based on the connected graph G. Then C can be extended to a graphical code C^* with parameters $[m, m - n + 2, g]$ provided that $n \geq 2g$. One may obtain such a code by adjoining to C any subgraph S with odd degree pattern W of weight $w \geq 2g$ as a further generator. Thus the resulting code consists of all even subgraphs of G together with all subgraphs with odd degree pattern W.* □

Example 10.11.11. The Petersen graph satisfies the condition of Lemma 10.11.10 with equality. In this simple case, an odd subgraph can be found by inspection; for instance, we may take the perfect matching of consisting of the five spoke edges, see Figure 10.9. Alternatively, we might as well use all of E, since all vertices have odd degree, namely 3.

Hence we may enlarge the Petersen code P to a $[15, 7, 5]$ code P^* by adjoining the complements of the subgraphs in P – that is, in terms of binary vectors, by taking the subspace of $V(15, 2)$ generated by P and the all-one vector. In view of Example 10.11.8, the weight enumerator of P^* is

$$A_{P^*}(x) \;=\; 1 + 18x^5 + 30x^6 + 15x^7 + 15x^8 + 30x^9 + 18x^{10} + x^{15}; \quad (10.19)$$

it would be a useful exercise for the reader to visualize the odd subgraphs in P^* for himself.

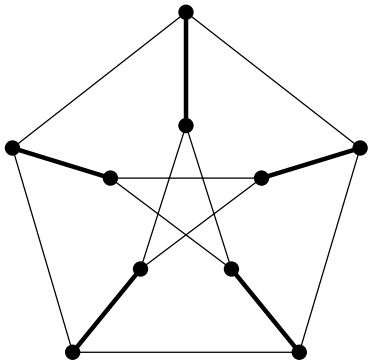


Fig. 10.9. A perfect matching of the Petersen graph

Example 10.11.12. The complete graph on six vertices leads to a code C with parameters $[15, 10, 3]$; see Example 10.11.6. Again, Lemma 10.11.10 applies, and we may augment this code to a $[15, 11, 3]$ code C^*.

It is now natural to try to go on augmenting the even graphical codes by more than one dimension (say, by k dimensions). Again, Lemma 10.11.10 gives the key ingredient needed. Clearly, what is required is a suitable collection of k linearly independent subgraphs $S_1, \ldots, S_k$ with odd degree patterns of weight $\geq 2g$ as further generators. In order to be compatible with each other, the sum of the odd degree patterns of any non-empty subset of these k subgraphs should also have weight at least $2g$, since their symmetric difference is a subgraph containing vertices of odd degree.

Hence, if we write C^* as the direct sum of C with the code D of dimension k generated by $S_1, \ldots, S_k$, the odd degree patterns of all non-empty subgraphs in D should have weight at least $2g$. In other words, the odd degree patterns associated with the words in D should themselves form an (even) binary code O with minimum distance $\geq 2g$; one calls O the *odd pattern code*. We have proved the following theorem likewise due to [JuVa97]:

Theorem 10.11.13. *Consider an even graphical code $C = C_E(G)$ with parameters $[m, m - n + 1, g]$ based on the connected graph G. Then C can be extended to a graphical code C^* with parameters $[m, m-n+k+1, g]$ provided that there exists an even binary $[n, k, 2g]$ code O. One may obtain such a code by adjoining to C any k linearly independent subgraphs $S_1, \ldots, S_k$ with odd degree patterns forming a basis for O as further generators.* □

Example 10.11.14. Let G be the complete graph on 16 vertices; then $C = C_E(G)$ is a binary $[120, 105, 3]$ code, by Example 10.11.6. To apply Theorem 10.11.13, we require a (preferably large) even binary $[16, k, 6]$ code O.

We may obtain a suitable code by using the so-called *parity check extension* of the code P^* constructed in Example 10.11.11. This standard construction from coding theory proceeds as follows: we append a further coordinate to a given binary code D with odd minimum distance, in this case to $D = P^*$. Then we extend the codewords of D by appending 0 to words of even weight, and 1 to words of odd weight; thus the extended codewords all have even weight. Note that this construction keeps the dimension of D, adds 1 to the length, and raises the minimum distance by 1.

The code O constructed in this way from P^* has parameters $[16, 7, 6]$. Hence Theorem 10.11.13 implies the existence of a graphical $[120, 112, 3]$ code C^*. This example shows how graphical codes may be constructed recursively by using smaller graphical codes (or parity check extensions of such codes) for the required odd pattern code O.

It is not difficult to realize that a better graphical augmentation than that guaranteed by Theorem 10.11.13 may be possible if one makes use of the precise structure of G. For example, if an odd degree pattern W of weight w is actually an independent subset of G, we obviously need at least w edges to cover the w vertices in W (and to produce the given odd degree pattern W). This idea leads to the following result also proved in [JuVa97]; we shall leave the details to the reader.

Theorem 10.11.15. *Consider an even graphical code $C = C_E(G)$ with parameters $[m, m - n + 1, g]$ based on the connected graph G and assume that $V_1, \ldots, V_c$ is a partition of the vertex set V into independent sets with cardinalities $n_1, \ldots, n_c$, respectively. Then C can be extended to a graphical code C^* with parameters $[m, m-n+1+(k_1+\ldots+k_c), g]$ provided that there exist even binary $[n_i, k_i, g]$ codes O_i for $i = 1, \ldots, c$. One may obtain such a code C^* by adjoining to C (for $i = 1, \ldots, c$) arbitrary sets of k_i linearly independent subgraphs of G with odd degree patterns contained in V_i and forming a basis for O_i (when considered as binary vectors of length n_i indexed with the vertices in V_i) as further generators.* □

The main problem for applying Theorem 10.11.15 consists in finding a suitable partition of the vertex set of G into independent sets. This may, of course, be viewed as a coloring problem: color V in such a way that no edge has both its end vertices colored with the same color. This requirement severely restricts the applicability of Theorem 10.11.15, since it is well-known that determining an optimal such partition – that is, finding the chromatic number $\chi(G)$ of G – is an NP-hard problem for $\chi(G) \geq 3$; see [Kar72]. On the other hand, the graphs with chromatic number 2 are just the bipartite graphs, by Example 9.1.1; also, finding a bipartition is easy, by Theorem 3.3.7. Accordingly, Theorem 10.11.15 is mainly useful for augmenting even graphical codes based on bipartite graphs (which, of course, implies an even value of g).

Let us give an example related to a famous class of codes, the so-called *extended binary Hamming codes*. These are codes with parameters of the form

$[2^h, 2^h - h - 1, 4]$; they exist for every integer $h \geq 2$ and are actually uniquely determined by their parameters (up to a permutation of the coordinate positions); moreover, they are the largest possible codes for the given length and minimum distance 4.

Example 10.11.16. Consider the complete bipartite graph $G = K_{p,p}$, where $p = 2^a$. By Theorem 10.11.5, $C = C_E(G)$ is a $[2^{2a}, 2^{2a} - 2^{a+1} + 1, 4]$ code. We apply Theorem 10.11.15 with $c = 2$ and $n_1 = n_2 = 2^a$. Then we may use for both O_1 and O_2 the extended binary Hamming code with parameters $[2^a, 2^a - a - 1, 4]$, resulting in a graphical code C^* with parameters $[2^{2a}, 2^{2a} - 2a - 1, 4]$. By the remark above, C^* is in fact the extended binary Hamming code with these parameters. Hence the extended binary Hamming code with parameters $[2^h, 2^h - h - 1, 4]$ is graphical for all even values of h.

Exercise 10.11.17. Prove that the extended binary Hamming code with parameters $[2^h, 2^h - h - 1, 4]$ is graphical also for all odd values of $h \geq 5$, by using a similar construction as in Example 10.11.16.

We remark that the constructions given in Theorems 10.11.13 and 10.11.15 generalize earlier constructions proposed by Hakimi and Bredeson [HaBr68]. However, it is not at all obvious from their paper whether or not the codes constructed there are graphical; to see this requires some work. It turns out that the methods of [HaBr68] produce exactly those codes which can be obtained by recursive applications of Theorems 10.11.13 and 10.11.15 when the odd pattern codes are restricted to even graphical codes and to smaller graphical codes already constructed in this way; see [JuVa97] who have called such codes *purely graphical*. In this context, we mention another interesting result obtained in [JuVa97], which actually provides an alternative construction for the extended binary Hamming codes.

Exercise 10.11.18. Prove that the extended binary Hamming codes are purely graphical codes. Hint: Use Example 10.11.16 and Exercise 10.11.17 for a recursive construction of codes with the parameters in question.

The preceding result is some indication that graphical codes can have good parameters; more evidence for this claim is given in [JuVa97]. Of course, one needs suitable graphs to start the construction. Again, it is no trivial task to find a graph on a desired number of edges with a large girth. For instance, it is known that in a graph with minimum degree $\delta \geq 3$, the girth can only be logarithmic in the number of vertices; we refer the reader to [Bol78, Section III.1] for more information on this topic. Hence one should usually start with graphs having (many) vertices of degree two. As an illuminating exercise, the reader might try to devise a suitable construction method for such graphs; an algorithm solving this problem may be found in [Hak66].

There is a further problem that needs to be addressed: if graphical codes are ever to become practically competitive, one certainly needs an efficient

encoding and decoding procedure. It is not enough to just have a good code, one needs to be able to actually do the necessary computations! As always with linear codes, encoding can be done easily, using a so-called generator matrix; we refer the reader to the text books on coding theory already cited. So the crucial question is how to decode a graphical code. We note that decoding an arbitrary linear code is an NP-hard problem, by a result of [BeMT94]. Fortunately, algorithmic graph theory and combinatorial optimization can also be used to solve this problem for graphical codes. We will postpone this to Chapter 14, as we do net yet have the relevant notions at our disposal.

Let us remark that the relationship between graph theory and coding theory discussed in the present section is not a one-way street: it is also possible to use this approach to obtain interesting results about graphs. By viewing the cycle space and the bond space of a graph as binary linear codes, one can obtain a very short and elegant proof of Read's theorem [Rea62] giving the generating function for the number of Eulerian graphs with n vertices and an analogous theorem concerning bipartite Eulerian graphs; see [JuVa95].

We conclude this section by noting that a similar approach can also be used to construct non-binary linear codes. This is rather natural for the ternary case: one may replace the graph G by a connected (but not necessarily strongly connected) digraph and the even subgraphs of G by the ternary circulations on that digraph. It is less obvious that this approach can also be transferred to the q-ary case. Again, these ideas basically go back to Bredeson and Hakimi [HaBr69, BoHa71]. Later, Jungnickel and Vanstone extended their approach from [JuVa97] also to the ternary and the q-ary case; see [JuVa99a, JuVa99b].

11

The Network Simplex Algorithm

O sancta simplicitas!
John Huss

For practical applications, by far the most useful optimization algorithm for solving linear programs is the celebrated simplex algorithm. With professional implementation it has a remarkable performance: problems with ≈1000 variables and ≈1000 restrictions can be dealt with within 0.1 to 0.5 seconds. This suggests trying to apply this algorithm also to problems from graph theory. Indeed, the most important network optimization problems may be formulated in terms of linear programs; this holds, for instance, for the determination of shortest paths, maximal flows, optimal flows, and optimal circulations.

Nevertheless, a direct application of the usual simplex algorithm would make no sense, as the resulting linear programs would be unwieldy and highly degenerate. These two problems are avoided by using a suitable graph theoretic specialization of the simplex algorithm, the *network simplex algorithm*. This algorithm is usually formulated in terms of a standard problem which we will introduce in the first section, namely the *minimum cost flow problem*; all other problems of practical interest admit easy transformations to this problem.

For a long time, the existence of a provably efficient version of the network simplex algorithm was one of the major open problems in complexity theory, even though it was clearly the most efficient practical algorithm for the minimum cost flow problem. This problem was finally solved by Orlin [Orl97] who gave an implementation with complexity $O(|V|^2|E|\log(|V|C))$, where $C = \max\{|\gamma(e)|: e \in E\}$ is the maximum of the cost values appearing. An improved complexity bound of $O(|V||E|\log|V|\log(|V|C))$ was achieved in [Tar97]. For more background, we also mention the books [Chv83] and [BaJS90] as well as the papers [GoHa90], [GoHa91], and [GoGT91]; in these papers, suitable dual versions of the network simplex algorithm were shown to have polynomial running time – something of a breakthrough.

In this book, we have decided to emphasize the graph theoretical aspects of combinatorial optimization while avoiding the theory of linear programming as much as possible. In view of this philosophy, it is very fortunate that the network simplex algorithm may be dealt with entirely in graph theoretic

terms, with no need to appeal to linear programming. This will be done in the present chapter, using many of the ideas and concepts we have already met. Nevertheless, a working knowledge of the ordinary simplex algorithm would, of course, be helpful, as it would provide additional motivation for the notions to be introduced in this chapter.

11.1 The minimum cost flow problem

The minimum cost flow problem (MCFP) is arguably the most fundamental among the flow and circulation problems, as all such problems may be transformed easily to the MCFP, and as this problem can be solved extremely efficiently using the network simplex algorithm. The MCFP is a common generalization of the transshipment problem defined in Example 10.10.1 (where additional lower capacity restrictions are added) and the minimum cost circulation problem studied extensively in Chapter 10 (where the flow conservation condition is replaced by a demand condition).

Definition 11.1.1 (minimum cost flow problem). Let $G = (V, E)$ be a connected digraph, and let the following data be given:

- *upper* and *lower capacity functions* $b\colon E \to \mathbb{R}$ and $c\colon E \to \mathbb{R}$, respectively;
- a *cost function* $\gamma\colon E \to \mathbb{R}$;
- a *demand function* $d\colon V \to \mathbb{R}$ with $\sum_{v\in V} d(v) = 0$.

The *minimum cost flow problem* (MCFP) requires the determination of a mapping $f\colon E \to \mathbb{R}$ with minimal cost $\gamma(f) = \sum_{e\in E} \gamma(e)f(e)$ subject to the following two conditions:

(F1) $b(e) \le f(e) \le c(e)$ for all $e \in E$ (*capacity restrictions*);

(F2) $\sum_{e^+=v} f(e) - \sum_{e^-=v} f(e) = d(v)$ for all $v \in V$ (*demand restrictions*).

Vertices with a negative demand (which we might also view as a *supply*, as in the supply and demand problem studied in Section 7.7) are called *sources*, and vertices with a positive demand are referred to as *sinks*; all other vertices may again be considered as *transshipment nodes*. A *flow* is a map $f\colon E \to \mathbb{R}$ satisfying the demand restrictions for all $v \in V$; if, in addition, the capacity restrictions hold for all $e \in E$, one speaks of an *admissible flow*. Thus the MCFP asks for an admissible flow of minimum cost.

A small example of an MCFP is displayed in Figure 11.1; and in Table 11.1 we indicate how some other standard problems can be transformed into MCFP's.

Before we can try to find a flow of minimum cost, we have to decide if there are admissible flows at all. To this purpose, we shall generalize the supply and demand theorem 7.7.1 to the case where a lower capacity function appears;

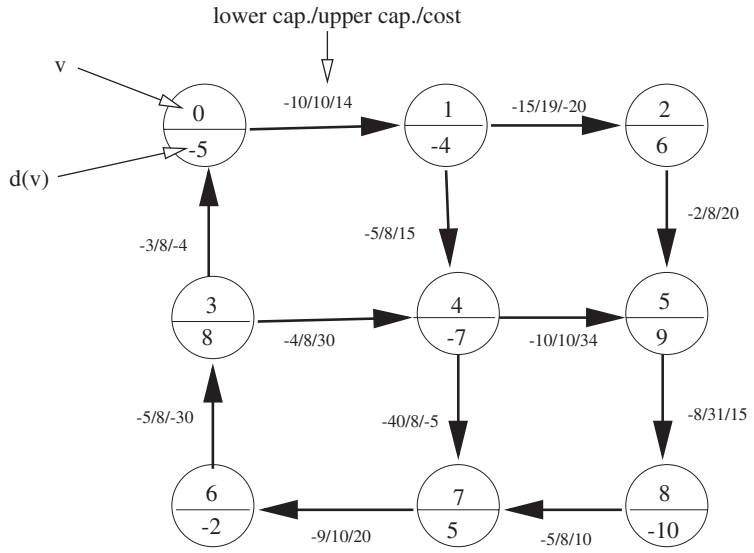


Fig. 11.1. A minimum cost flow problem

Table 11.1. Some problem transformations

Problem	**Specifications**
Shortest path from s to t with respect to lengths w	$b \equiv 0;\ c \equiv 1;\ \gamma = w;\ d(s) = -1;\ d(t) = 1;$ $d(v) = 0$ for $v \neq s, t$
Maximum flow for the flow network $N = (G, c_1, s, t)$	$b \equiv 0;\ c = c_1;\ d(v) = 0$ for $v \in V;\ \gamma(e) = 0$ for $e \in E;$ new *return arc* ts with $\gamma(ts) = -1,$ $b(ts) = 0,\ c(ts) = \sum_{e^- = s} c_1(e)$
circulation problem	$d(v) = 0$ for $v \in V;$ remaining data as given

this is similar to the transformation used in Section 10.2 to decide if there are feasible circulations.

As in the context of circulations, a *cut* will just be a partition $V = S \dot\cup T$, and the *capacity* of such a cut is

$$c(S,T) \ = \sum_{e^- \in S, e^+ \in T} c(e) - \sum_{e^+ \in S, e^- \in T} b(e).$$

We can now prove the following common generalization of Theorems 7.7.1 and 10.2.7.

Theorem 11.1.2. *An MCFP P as in Definition 11.1.1 allows an admissible flow if and only if the following condition holds for every cut $V = S \dot\cup T$:*

$$c(S,T) \;\geq\; \sum_{v\in T} d(v). \tag{11.1}$$

Proof. By Theorem 7.7.1, the assertion holds provided that $b(e) = 0$ for all $e \in E$. If this is not the case, we put

$$\begin{aligned} b'(e) \;&=\; 0 \qquad \text{for } e \in E,\\ c'(e) \;&=\; c(e) - b(e) \qquad \text{for } e \in E,\\ d'(v) \;&=\; d(v) + \sum_{e^-=v} b(e) - \sum_{e^+=v} b(e) \qquad \text{for } v \in V, \end{aligned}$$

and denote the resulting problem by P'; the cost function is immaterial in this context. One easily checks that a mapping f is an admissible flow for P if and only if the mapping g defined by

$$g(e) \;=\; f(e) - b(e) \qquad \text{for } e \in E$$

is an admissible flow for P'. By Theorem 7.7.1, there is an admissible flow for P' – and hence also for P – if and only if the condition

$$c'(S,T) \;\geq\; \sum_{v\in T} d'(v) \tag{11.2}$$

holds for every cut (S,T). But

$$c'(S,T) \;=\; \sum_{e^-\in S, e^+\in T} (c(e) - b(e)) = c(S,T) + \sum_{e^+\in S, e^-\in T} b(e) - \sum_{e^-\in S, e^+\in T} b(e)$$

and

$$\sum_{v\in T} d'(v) \;=\; \sum_{v\in T} d(v) + \sum_{e^-\in T, e^+\in S} b(e) - \sum_{e^+\in T, e^-\in S} b(e).$$

Because of these two equalities, (11.1) holds if and only if (11.2) holds. □

11.2 Tree solutions

Consider an MCFP P on a digraph G, and let T be a spanning tree for G. An admissible flow f is called a *tree solution* for P (with respect to T) if the flow value $f(e)$ is either the lower capacity $b(e)$ or the upper capacity $c(e)$ of e, whenever e is an edge outside of T. As we will see, the existence of an admissible flow for P implies the existence of an optimal tree solution for P. Moreover, a non-optimal tree solution may always be improved to a better tree solution by exchanging just one edge, as in Section 4.3. The network simplex

algorithm uses operations of this type until an optimal tree solution is reached. These exchange operations are much simpler and may also be implemented much more efficiently than the cycle cancelling operations used in Chapter 10; this is the major advantage of the network simplex algorithm.

Now let f be an admissible flow for P. We call an edge e *free* (with respect to f) provided that $b(e) < f(e) < c(e)$. Thus f is a tree solution for P if and only if there is some spanning tree T containing all the free edges. It should be noted that we do not require that all edges of T be free: an admissible flow f may be a tree solution with respect to different spanning trees. We now prove the following fundamental result already mentioned above.

Theorem 11.2.1. *Let P be an MCFP on a digraph G, as in Definition 11.1.1. If P allows an admissible flow, then it also admits an optimal tree solution.*

Proof. We use a continuity argument as in the proof of Theorem 6.1.6. As the set of admissible flows for an MCFP is a compact subset of $\mathbb{R}^{|E|}$ and as the cost function is continuous, there exists some optimal solution f. We may assume that the free edges with respect to f are not contained in a spanning tree for G; otherwise, there is nothing to prove. Then there exists an (undirected) cycle C in G consisting of free edges only; hence we may augment f by cancelling this cycle; it is immaterial which of the two possible orientations of C we use. (Cancelling a cycle is, of course, done exactly as in Chapter 10 and formally described in Algorithm 10.4.1.) As in the case of circulations, the cost of both orientations of C clearly has to be nonnegative (and then actually 0), because of the optimality of f. After cancelling C, at least one edge reaches either its lower or upper capacity bound, and therefore is no longer free with respect to the resulting optimal flow f'. Hence this operation decreases the number of free edges by at least one. Continuing in this way, we will eventually reach an optimal flow g such that the set F of free edges with respect to g does not contain any cycle. As G is connected, F is contained in a spanning tree for G, and hence g is an (optimal) tree solution. □

Now let T be some spanning tree for G. In general, there will be many tree solutions with respect to T, as each edge in $E \setminus T$ may reach either its lower or its upper capacity. Indeed, we can obtain a candidate for a tree solution by prescribing a lower or upper capacity for each $e \in E \setminus T$:

Lemma 11.2.2. *Consider an MCFP P on a digraph G, let T be a spanning tree for G, and let $E \setminus T = L \dot\cup U$ be any partition of $E \setminus T$. Then there exists a unique flow f satisfying $f(e) = b(e)$ for all $e \in L$ and $f(e) = c(e)$ for all $e \in U$.*

Proof. Put

$$f(e) = b(e) \text{ for all } e \in L \quad \text{and} \quad f(e) = c(e) \text{ for all } e \in U. \qquad (*)$$

Now let v be any leaf of T, and e the unique edge of T incident with v. Then the demand restriction for v together with $(*)$ uniquely determines the flow

value $f(e)$. The same reasoning may then be applied to each leaf of the tree $T \setminus e$ etc. In this way, one recursively defines the flow values $f(e)$ for all edges $e \in T$ while simultaneously satisfying all demand restrictions. □

One calls a partition $L \dot\cup U$ of $E \setminus T$ a *tree structure*. Note that the flow constructed in Lemma 11.2.2 (for a prescribed tree structure (T, L, U)) is not necessarily admissible. Thus it makes sense to call (T, L, U) *admissible* or *optimal* if the associated flow has the respective property.

We now prove an optimality criterion which will allow us to decide if a tree solution is already optimal. Recall that a *potential* is just a map $\pi : V \to \mathbb{R}$, and that the associated *reduced cost function* is defined by

$$\gamma_\pi(uv) \;=\; \gamma(uv) + \pi(u) - \pi(v) \quad \text{for } uv \in E.$$

Theorem 11.2.3. *An admissible tree structure (T, L, U) for an MCFP is optimal if and only if there exists a potential $\pi : V \to \mathbb{R}$ satisfying*

$$\gamma_\pi(e) \begin{cases} = 0 & \text{for all } e \in T, \\ \geq 0 & \text{for all } e \in L, \\ \leq 0 & \text{for all } e \in U. \end{cases} \tag{11.3}$$

Proof. Let $g : E \to \mathbb{R}$ be any flow. Then

$$\begin{aligned} \gamma_\pi(g) \;&=\; \sum_{uv \in E} g(uv)(\gamma(uv) + \pi(u) - \pi(v)) \\ &=\; \sum_{uv \in E} g(uv)\gamma(uv) + \sum_{u \in V} \sum_{e^- = u} g(e)\pi(u) - \sum_{v \in V} \sum_{e^+ = v} g(e)\pi(v) \\ &=\; \gamma(g) + \sum_{v \in V} \pi(v)\Big(\sum_{e^- = v} g(e) - \sum_{e^+ = v} g(e)\Big) \\ &=\; \gamma(g) - \sum_{v \in V} \pi(v)d(v), \end{aligned}$$

where we have used the demand restrictions in the final step. Thus $\gamma_\pi(g)$ and $\gamma(g)$ differ only by a constant which is independent of g. Hence an admissible flow is optimal for the cost function γ if and only if it is optimal for γ_π.

Let $f : E \to \mathbb{R}$ be the admissible flow defined by an admissible tree structure (T, L, U) satisfying (11.3). By definition,

$$f(e) \;=\; b(e) \text{ for all } e \in L \quad \text{and} \quad f(e) \;=\; c(e) \text{ for all } e \in U. \tag{11.4}$$

Now consider an admissible flow g. By (11.3) and (11.4),

$$\begin{aligned} \gamma_\pi(g) \;&=\; \sum_{e \in T} g(e)\gamma_\pi(e) + \sum_{e \in L} g(e)\gamma_\pi(e) + \sum_{e \in U} g(e)\gamma_\pi(e) \\ &=\; \sum_{e \in L} g(e)\gamma_\pi(e) + \sum_{e \in U} g(e)\gamma_\pi(e) \end{aligned}$$

$$\geq \sum_{e\in L} b(e)\gamma_\pi(e) + \sum_{e\in U} c(e)\gamma_\pi(e)$$
$$= \sum_{e\in E} f(e)\gamma_\pi(e) \;=\; \gamma_\pi(f).$$

Thus f is optimal for the reduced cost function γ_π and hence also for γ. □

The potential in the optimality condition (11.3) has the property that all edges of some spanning tree have reduced cost 0. The network simplex algorithm uses potentials of this kind only. We next show that any spanning tree T determines such a potential, which is unique up to prescribing one value.

Lemma 11.2.4. *Let (T, L, U) be a tree structure for an MCFP, and let x be any vertex. Then there exists a unique potential π satisfying*

$$\pi(x) = 0 \quad \textit{and} \quad \gamma_\pi(e) = 0 \;\; \textit{for all } e \in T. \tag{11.5}$$

Proof. More explicitly, condition (11.5) requires

$$\gamma(uv) + \pi(u) - \pi(v) = 0 \quad \text{for each edge } e = uv \in T \quad (*).$$

Because of $\pi(x) = 0$, the values $\pi(r)$ are determined by $(*)$ for every vertex r adjacent to x. In the same way, π is then also determined for the neighbors of these vertices etc. As T contains a unique path from x to every other vertex of G, condition $(*)$ indeed yields a unique potential π with $\pi(x) = 0$. □

11.3 Constructing an admissible tree structure

To start the network simplex algorithm, we need some admissible tree structure for the given MCFP. This can be achieved by a suitable transformation.

Theorem 11.3.1. *Consider an MCFP P on a digraph $G = (V, E)$ as in 11.1.1, and let the auxiliary MCFP P' be given by the data in Table 11.2.*[1] *Then there exists an optimal solution g for P'. Moreover, we have one of the following two alternatives:*

- *If $g(xt) > 0$ for some $xt \in E'$, then there exists no admissible flow for P.*
- *If $g(xv) = 0$ for all $xv \in E'$, then the restriction f of g to E is an optimal flow for P.*

Proof. The map h defined by $h(e) = b(e)$ for $e \in E$ and $h(e) = c'(e) - 1$ for $e \in E' \setminus E$ obviously satisfies the capacity restrictions for P'. We check that

[1] The purpose of the constants +1 appearing in the definition of c' is to make the auxiliary problem nondegenerate.

Table 11.2. The auxiliary problem P'

$V' = V \cup \{x\}$ (with $x \notin V$)
$d'(v) = d(v)$ for $v \in V$; $d'(x) = 0$
$E' = E \cup \{xv : d(v) + \sum_{e^+=v} b(e) - \sum_{e^-=v} b(e) < 0\}$ $\cup \ \{vx : d(v) + \sum_{e^+=v} b(e) - \sum_{e^-=v} b(e) \geq 0\}$
$b'(e) = b(e)$ for $e \in E$; $b'(xv) = 0$ for $xv \in E'$; $b'(vx) = 0$ for $vx \in E'$
$c'(e) = c(e)$ for $e \in E$; $c'(xv) = d(v) - \sum_{e^+=v} b(e) + \sum_{e^-=v} b(e) + 1$ for $xv \in E'$; $c'(vx) = -d(v) + \sum_{e^+=v} b(e) - \sum_{e^-=v} b(e) + 1$ for $vx \in E'$
$\gamma'(e) = \gamma(e)$ for $e \in E$; $\gamma'(xv) = M$ for $xv \in E'$; $\gamma'(vx) = M$ for $vx \in E'$, where $M := 1 + \frac{1}{2}\lvert V\rvert \max\{\lvert\gamma(e)\rvert : e \in E\}$

the demand restrictions are likewise satisfied. Let v be an arbitrary vertex in V, and assume first $xv \in E'$. Then

$$\begin{aligned}\sum_{\substack{e\in E'\\ e^+=v}} h(e) - \sum_{\substack{e\in E'\\ e^-=v}} h(e) &= h(xv) + \sum_{\substack{e\in E\\ e^+=v}} b(e) - \sum_{\substack{e\in E\\ e^-=v}} b(e)\\ &= c'(xv) - 1 + \sum_{\substack{e\in E\\ e^+=v}} b(e) - \sum_{\substack{e\in E\\ e^-=v}} b(e)\\ &= d(v) = d'(v).\end{aligned}$$

Now assume $vx \in E'$. Then

$$\begin{aligned}\sum_{\substack{e\in E'\\ e^+=v}} h(e) - \sum_{\substack{e\in E'\\ e^-=v}} h(e) &= -h(vx) + \sum_{\substack{e\in E\\ e^+=v}} b(e) - \sum_{\substack{e\in E\\ e^-=v}} b(e)\\ &= -c'(vx) + 1 + \sum_{\substack{e\in E\\ e^+=v}} b(e) - \sum_{\substack{e\in E\\ e^-=v}} b(e)\\ &= d(v) = d'(v).\end{aligned}$$

This proves the validity of the demand restrictions for all vertices in V. It remains to check the new vertex x:

$$\begin{aligned}\sum_{\substack{e\in E'\\ e^+=x}} h(e) - \sum_{\substack{e\in E'\\ e^-=x}} h(e) &= \sum_{vx\in E'} (c'(vx) - 1) - \sum_{xv\in E'} (c'(xv) - 1)\\ &= \sum_{v\in V} \left(-d(v) - \sum_{\substack{e\in E\\ e^+=v}} b(e) + \sum_{\substack{e\in E\\ e^-=v}} b(e) \right)\end{aligned}$$

$$\begin{aligned} &= -\sum_{v\in V} d(v) - \sum_{e\in E} b(e) + \sum_{e\in E} b(e) \\ &= 0 \;=\; d'(x). \end{aligned}$$

Hence h is an admissible flow for P', and by Theorem 11.2.1 P' admits an optimal flow, in fact even an optimal tree solution. Let us choose any optimal flow g.

Case 1. $g(xt) > 0$ for some $xt \in E'$.

We first claim that no cycle C in P' can contain two edges xv and wx satisfying $g(xv) > 0$ and $g(wx) > 0$, and such that one may cancel C, where we use the orientation of C opposite to that of xv. Assume otherwise. Note that C contains at most $|V| - 1$ edges besides xv and wx and that

$$-2M + (|V| - 1)\max\{|\gamma(e)|\colon e \in E\} \;<\; -2M + 2M \;=\; 0.$$

Therefore augmenting g by cancelling C would strictly decrease the cost, which contradicts the optimality of g. This proves our auxiliary claim.

Now let us assume, by way of contradiction, that there exists an admissible flow for P. Then every cut $V = S \,\dot\cup\, T$ satisfies

$$c(S,T) \;\geq\; \sum_{v\in T} d(v), \tag{11.6}$$

by Theorem 11.1.2. Choose a vertex t with $g(xt) > 0$, and let T denote the set of all vertices $v \in V$ for which one may be reach t via an augmenting path for g.[2] Assume $g(vx) > 0$ for some $v \in T$. Then we can construct a cycle through xt and vx along which we may augment in the opposite direction of xt: an augmenting path from v to t, followed by the backward edges xt and vx. This contradicts our auxiliary claim above, and we conclude

$$g(vx) = 0 \;\text{ or }\; vx \notin E' \quad \text{for all } v \in T. \tag{11.7}$$

Now put $S = V \setminus T$. Then

$$g(e) \;=\; \begin{cases} c(e) & \text{if } e^- \in S,\; e^+ \in T, \\ b(e) & \text{if } e^+ \in S,\; e^- \in T; \end{cases} \tag{11.8}$$

for otherwise we could reach t via an augmenting path from either $e^- \in S$ or $e^+ \in S$, contradicting the definition of T. Equation (11.8) implies

$$\sum_{e^+\in T} g(e) - \sum_{e^-\in T} g(e) \;=\; \sum_{e^-\in S, e^+\in T} g(e) - \sum_{e^+\in S, e^-\in T} g(e)$$

[2] By analogy with Chapter 6, a path W from v to t is called an *augmenting path* with respect to g if $g(e) < c(e)$ holds for every forward edge $e \in W$, and $g(e) > b(e)$ holds for every backward edge $e \in W$.

$$\begin{aligned} &= \sum_{e^- \in S, e^+ \in T} c(e) - \sum_{e^+ \in S, e^- \in T} b(e) \\ &= c(S,T). \end{aligned}$$

Let us put $g(xv) = 0$ if $xv \notin E'$, and $g(vx) = 0$ if $vx \notin E'$. Using the preceding equation together with (11.7) and $g(xt) > 0$, we compute

$$\begin{aligned} \sum_{v \in T} d(v) &= \sum_{v \in T} \left(\sum_{e \in E', e^+ = v} g(e) - \sum_{e \in E', e^- = v} g(e) \right) \\ &= \sum_{v \in T} \left(g(xv) - g(vx) + \sum_{e \in E, e^+ = v} g(e) - \sum_{e \in E, e^- = v} g(e) \right) \\ &= \sum_{v \in T} g(xv) + \sum_{e \in E, e^+ \in T} g(e) - \sum_{e \in E, e^- \in T} g(e) \\ &= \sum_{v \in T} g(xv) + c(S,T) \;>\; c(S,T), \end{aligned}$$

contradicting (11.6). Thus P does not allow an admissible flow, which proves the assertion in Case 1.

Case 2. $g(xv) = 0$ for all $xv \in E'$.

As $d'(x) = 0$, we must also have $g(vx) = 0$ for all $vx \in E'$. Thus the restriction f of g to E is an admissible flow for P. If P were also to admit a flow f' with strictly smaller cost, we could extend f' to an admissible flow for P' by putting $f'(e) = 0$ for all $e \in E' \setminus E$; but this flow would have smaller cost than the optimal flow g, a contradiction. Hence f is indeed an optimal flow for P. □

As we have seen, the auxiliary problem P' always allows an admissible flow and hence, by Theorem 11.2.1, also an admissible tree structure. We shall now exhibit one such structure explicitly.

Lemma 11.3.2. *Let P' be the auxiliary MCFP given by the data in Table 11.2, and put*

$$T \;=\; \{xv \colon xv \in E'\} \cup \{vx \colon vx \in E'\}, \;\; L \;=\; E, \;\; \text{and } U \;=\; \emptyset.$$

Then (T, L, U) is an admissible tree structure for P'.

Proof. Since x is joined to each vertex in V by exactly one edge, T is indeed a spanning tree for $G' = (V', E')$. Let g be the flow associated with (T, L, U) according to Lemma 11.2.2. Then $g(e) = b(e)$ for all $e \in E$. Now the demand restrictions determine the values $g(xv)$ and $g(vx)$ uniquely, and thus g agrees with the admissible flow defined in the first part of the proof of Theorem 11.3.1. Hence (T, L, U) is indeed admissible for P'. □

Exercise 11.3.3. Fill in the details of the proof of Lemma 11.3.2 and show that g indeed agrees with the tree solution associated with (T, L, U).

It should also be noted that it is not absolutely necessary to introduce the auxiliary problem P': there are other ways to determine an admissible tree structure for P.[3] One first constructs an arbitrary feasible solution; as in the special case of circulations, this can be done efficiently. Then one cancels free cycles as long as possible, and finally uses the remaining free edges together with suitable other edges to determine an admissible tree solution. In my group at the University of Augsburg, we have implemented such an algorithm in the free software package GOBLIN which treats many fundamental optimization problems for graphs and networks; GOBLIN is available from the URL

`http://www.math.uni-augsburg.de/opt/goblin.html`

Another implementation of the network simplex algorithm which may be obtained free of charge for academic use is the MCFZIB-code; see

`http://www.zib.de/Optimization/Software/Mcf/`

11.4 The algorithm

We can now describe the general structure of the network simplex algorithm, though we will have to be more specific later to ensure termination. It will simplify matters to deviate from our usual way of writing algorithms and just give a very concise description split into appropriate blocks.

Algorithm 11.4.1 (network simplex algorithm). Let P be an MCFP on a digraph G as in Definition 11.1.1, and let P' be the associated auxiliary MCFP P' with the data given in Table 11.2.

1. Initialization. Put

$$T = E' \setminus E; \quad L = E; \quad U = \emptyset;$$

$$f(e) = b(e) \text{ for } e \in E \quad \text{and} \quad f(e) = c'(e) - 1 \text{ for } e \in E' \setminus E;$$

$$\pi(x) = 0; \quad \pi(v) = M \text{ for } v \in V \text{ with } xv \in E'; \quad \text{and}$$
$$\pi(v) = -M \text{ for } v \in V \text{ with } vx \in E'.$$

[3] The approach of Theorem 11.3.1 corresponds to the *big-M method* in linear programming. There is a major difference, though: for an MCFP, we can explicitly select a reasonable value of M, whereas for general linear programs M has to be taken really huge – which automatically leads to numerical problems. For instance, with $M = 10^{16}$ and a standard floating point arithmetic accurate to 15 digits, computing a sum $M + x$ with $0 < x < 10$ has no meaning. In other words, the big-M method is of theoretical interest, but not suitable for practical use. In contrast, the method in Theorem 11.3.1 also works very well in practice. A value of M which is so large that it would lead to numerical instability could arise only if some cost values differ by $\approx 10^{15}/|V|$, which is not the case in practical applications.

2. Optimality test. If there is no $e \in L \cup U$ with either

$$e \in L \text{ and } \gamma_\pi(e) < 0 \quad \text{or} \quad e \in U \text{ and } \gamma_\pi(e) > 0 \quad (*)$$

stop: in case $f(e) > 0$ for some $e \in E' \setminus E$, there is no admissible flow for P; otherwise, the restriction f of g to E is an optimal flow for P.

3. Pricing. Choose some $e \in L \cup U$ satisfying $(*)$ and determine the unique cycle C contained in $T \cup \{e\}$.

4. Augmenting. Consider C as oriented in the direction of e if $e \in L$, and as oriented in the direction opposite to that of e if $e \in U$. Augment f (by an amount of δ) by cancelling C, so that at least one edge of C reaches either its upper or lower capacity bound. Choose such an edge a; here $a = e$ is only permissible in case $\delta > 0$.

5. Update. Put

$$\begin{aligned} T &= (T \setminus \{a\}) \cup \{e\}, \\ L &= \begin{cases} (L \setminus \{e\}) \cup \{a\} & \text{if } a \text{ reaches its lower capacity bound} \\ L \setminus \{e\} & \text{if } a \text{ reaches its upper capacity bound,} \end{cases} \\ U &= E' \setminus (T \cup L), \end{aligned}$$

and compute the unique potential π associated with (T, L, U) satisfying $\pi(x) = 0$, as outlined in the proof of Lemma 11.2.4. Go to Step 2.

Now consider an iteration taking place during the course of the algorithm. The arc e selected in Step 3 is called the *entering arc*, and the unique cycle $C \subset T \cup \{e\}$ the *associated pivot cycle*. The arc a selected in Step 4 is referred to as the *leaving arc*.

As in the case of the labelling algorithm or the algorithm of Klein, the generic version of the network simplex algorithm given above does not necessarily terminate. The reason behind this phenomenon is the possibility of *degenerate* tree structures: tree structures (T, L, U) where T does not consist of free edges only. In this case, a proper augmentation along the cycle determined in Step 3 may be impossible: we may have $\delta = 0$ in Step 4. Even though we do change the tree T (as the leaving arc a is distinct from the the entering arc e in this case), we might – after a series of updates – reach the same tree structure (T, L, U) again. One says that the algorithm may *cycle*. Indeed, this is a real danger, as it is not unusual in practical instances to have 90% of all tree structures degenerate. However, cycling can be prevented by choosing the leaving arc appropriately; fortunately, the extra effort required to do so even tends to speed up the algorithm by cutting down the number of iterations needed.

We shall now explain in detail how one may prevent cycling. To this end, we first choose a fixed vertex w which we will use as the root of all the spanning

trees constructed during the course of the algorithm. Then an admissible tree structure (T, L, U) will be called *strongly admissible* if, for every vertex $v \neq w$, the unique path from v to w in T is in fact an augmenting path for the tree solution f canonically associated with (T, L, U). In particular, an admissible tree structure (T, L, U) is strongly admissible provided that it is *nondegenerate*; that is, if all edges of T are free.

In the initialization stage (Step 1), we choose $w = x$. Then the initial tree structure $(T, L, U) = (E' \setminus E, E, \emptyset)$ is indeed strongly admissible for P': it is admissible by Theorem 11.3.2, and we have $g(xv) = c'(xv) - 1 > 0$ for $xv \in E'$ and $g(vx) = c'(vx) - 1 < c'(vx)$ for $vx \in E'$, so that T is nondegenerate.

Now consider an iteration taking place during the course of the algorithm. Let e be the entering arc selected in Step 3, and let $C \subset T \cup \{e\}$ be the associated pivot cycle. We consider C as oriented in the direction of e if $e \in L$, and as oriented in the direction opposite to that of e if $e \in U$. An arc in $C \setminus \{e\}$ will be called *blocking* if cancelling C results in a reaching either its upper or its lower capacity bound. The unique vertex of C which is closest to to the root w of T is called the *apex* of C. We will show that the following selection rule for the leaving arc preserves the strong admissibility of all tree structures occurring during the course of the algorithm.

Rule of the last blocking arc: *Starting with the apex of the pivot cycle C, traverse C according to its orientation and select the last blocking arc encountered as the leaving arc.*

To illustrate the rule of the last blocking arc in Figure 11.2, we have used the following notation:

- W: The path from the apex to the last blocking arc, following the orientation of C. In Figure 11.2, the apex is 2 and $W = 2 - 3 - 5$.
- W': The path from the apex to the last blocking arc, following the opposite orientation of C. In Figure 11.2, $W' = 2 - 4 - 6 - 8 - 10 - 9 - 7$.

Theorem 11.4.2. *Let (T, L, U) be a strongly admissible tree structure, and let $e \notin T$. If the leaving arc a is selected in Step 4 of Algorithm 11.4.1 according to the rule of the last blocking arc, then the resulting tree structure in Step 5 is again strongly admissible.*

Proof. Let f denote the tree solution associated with (T, L, U), and let g be the admissible flow which results by cancelling C according to Step 4. It remains to show that in the resulting spanning tree $T' = (T \cup \{e\}) \setminus \{a\}$ the path leading from v to w is augmenting with respect to g (for every vertex $v \neq w$). We shall distinguish four cases.

Case 1. v is the apex of C. Then the paths from v to w in T' and T agree, and the flow values have not been changed on the edges of this common path Q. As (T, L, U) was strongly admissible, Q is augmenting with respect to f and hence also with respect to g.

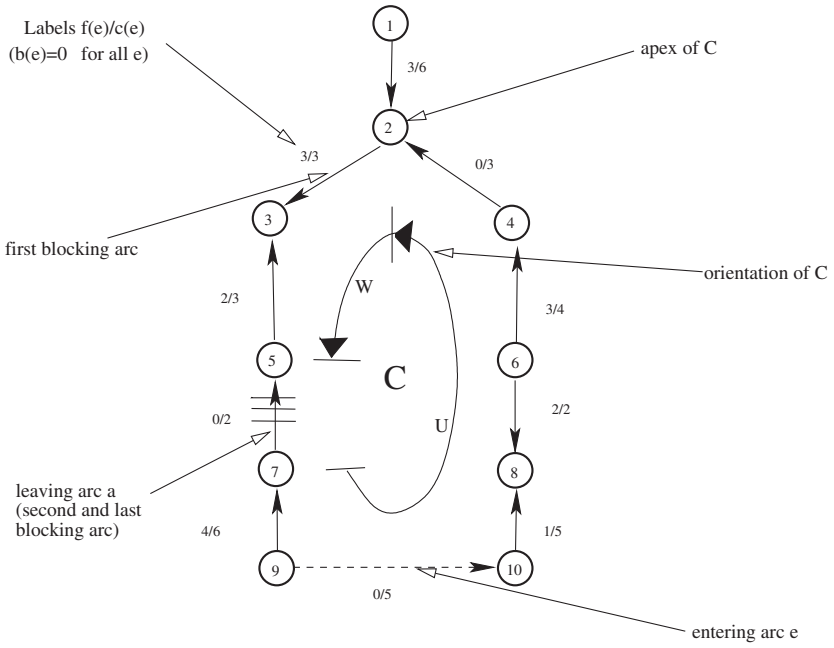


Fig. 11.2. The rule of the last blocking arc

Case 2. v is on W'. As a is the last blocking arc encountered by traversing C from its apex following its orientation, no edge in W' can be blocking. Therefore the path from v to the apex of C in T' is augmenting with respect to g. In view of Case 1, the path from v to w in T' is likewise augmenting with respect to g.

Case 3. v is on W. Let $\delta \geq 0$ be the amount of augmentation as in Step 4. In case $\delta > 0$, the flow f was increased on the edges of W by δ; hence the path opposite to W (which ends in the apex of C) is augmenting with respect to g. In view of Case 1, the path from v to w in T' is likewise augmenting with respect to g. Now assume $\delta = 0$. As a is not on W, the paths from v to w in T' and T agree, and the flow values have not been changed on the edges of this common path Q (because of $\delta = 0$). As (T, L, U) was strongly admissible, Q is augmenting with respect to f and hence also with respect to g.

Case 4. v is not on C. As (T, L, U) is strongly admissible, the path Z from v to w in T is augmenting with respect to f. Assume first that Z and C are disjoint. Then Z is also a path in T' which is augmenting also with respect to g, as the flow values have not changed on Z. Otherwise let y be the first vertex on Z which also belongs to C. Then the path from v to y in T' is augmenting

with respect to g (as before). By the previous three cases, the same conclusion holds for the path from y to w in T'. Hence the path from v to w in T' is likewise augmenting with respect to g. □

We need a further auxiliary result for proving that the network simplex algorithm terminates when using the rule of the last blocking arc.

Lemma 11.4.3. *Let (T, L, U) be a tree structure occurring during the course of Algorithm 11.4.1, and let π be the associated potential. Moreover, let a the leaving arc and $e = rs$ the entering arc, and denote the new potential (after augmenting and updating) by π'. Finally, let T_1 be the connected component of $T \setminus \{a\}$ containing w, and write $T_2 = V \setminus T_1$. Then*

$$\pi'(v) = \begin{cases} \pi(v) & \text{if } v \in T_1 \\ \pi(v) + \gamma_\pi(e) & \text{if } v \in T_2 \text{ and } r \in T_1 \\ \pi(v) - \gamma_\pi(e) & \text{if } v \in T_2 \text{ and } r \in T_2 \end{cases}$$

Proof. Write $T' = (T \cup \{e\}) \setminus \{a\}$. Then, by the definition of π',

$$\gamma(uv) + \pi'(u) - \pi'(v) = 0 \quad \text{for all } uv \in T'. \tag{11.9}$$

Let v be any vertex in T_1. Then the paths from v to w in T' and T agree, and from $\pi(w) = \pi'(w) = 0$ we conclude $\pi'(v) = \pi(v)$.

Now assume $r \in T_1$ and thus $s \in T_2$. Then $\pi'(r) = \pi(r)$ and hence $\pi'(s) = \gamma(rs) + \pi(r) = \pi(s) + \gamma_\pi(rs)$ by (11.9). Note that the paths from s to v in T' and T agree for every vertex $v \in T_2$. Therefore $\pi'(v) = \pi(v) + \gamma_\pi(rs)$ for all $v \in T_2$.

Finally, assume $r \in T_2$ and thus $s \in T_1$. Then $\pi'(s) = \pi(s)$ and therefore $\pi'(r) = -\gamma(rs) + \pi(s) = \pi(r) - \gamma_\pi(rs)$ by (11.9). Note that the paths from r to v in T' and T agree for every vertex $v \in T_2$. Therefore $\pi'(v) = \pi(v) - \gamma_\pi(rs)$ for all $v \in T_2$. □

In the following theorem, we assume that all data in our given MCFP are rational. Of course, this is no real restriction from a practical point of view, as we can represent only rational numbers in a computer anyway.

Theorem 11.4.4. *Let P be an MCFP on a digraph G as in Definition 11.1.1, and assume that all data b, c, d, γ are rational. Then the network simplex algorithm 11.4.1 terminates after finitely many steps provided that the rule of the last blocking arc is used for choosing the leaving arcs.*

Proof. By multiplying all data by their lowest common denominator, we may assume that the data are in fact integral. Then any augmentation by a positive value decreases the cost by at least 1 unit. But the cost of any admissible flow is bounded from below, so that there are only finitely many augmentations with $\delta > 0$. Hence it suffices to show that we can have only finitely many consecutive augmentations with $\delta = 0$.

Thus let us consider an augmentation with $\delta = 0$, starting with a strongly admissible tree structure (T, L, U). As usual, let C be the pivot cycle; $e = rs$ the entering and a the leaving arc; π the potential associated with (T, L, U), and π' the potential resulting after augmenting and updating. According to Step 3 in Algorithm 11.4.1, we distinguish two cases.

Case 1. $e \in L$ and $\gamma_\pi(e) < 0$. In this case, the orientation of e agrees with that of C. Then a lies between the apex of C and r when we traverse C according to its orientation. This is intuitively clear; see Figure 11.2. A formal proof can be given as follows. Let Z be the path in T from the end vertex s of e to the apex of C, following the orientation of C; in Figure 11.2, $Z = 10-8-6-4-2$. As (T, L, U) is strongly admissible, Z is augmenting with respect to the associated tree solution f. Because of $\delta = 0$, no arc on Z can be blocking. In particular, a indeed cannot lie on Z. This implies $r \in T_2$, and thus $\pi'(v) = \pi(v) - \gamma_\pi(rs) > \pi(v)$ for all $v \in T_2$, by Lemma 11.4.3.

Case 2. $e \in T$ and $\gamma_\pi(e) > 0$. In this case, the orientation of e is opposite to that of C. Now a lies between s and the apex of C when we traverse C according to its orientation; this is similar to the corresponding argument in the first case. Hence $r \in T_1$, and thus $\pi'(v) = \pi(v) + \gamma_\pi(rs) > \pi(v)$ for all $v \in T_2$, by Lemma 11.4.3.

Note $\pi'(v) = \pi(v)$ for all $v \in T_1$, again by Lemma 11.4.3. Hence the sum of all potentials $p(v)$ increases by at least 1 in both cases. As w always has potential 0, no potential can exceed $|V|C$, where $C = \max\{|\gamma'(e)| : e \in E'\}$. Hence the sum of all the potentials is bounded from above by $|V|^2 C$, and thus the number of consecutive augmentations with $\delta = 0$ is always finite. □

11.5 Efficient implementations

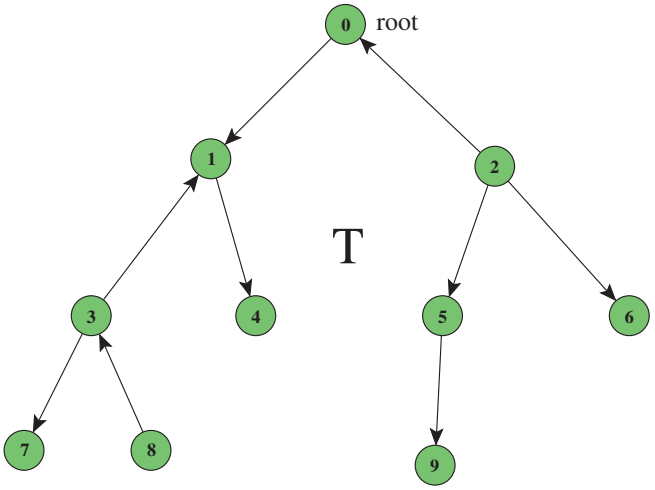
During each iteration of Algorithm 11.4.1, we need to compute the current pivot cycle C, augment along C, and update the potentials. In order to do all this efficiently, one requires information about the current spanning tree. One possibility for an efficient implementation uses so-called *tree indices* which we will now introduce.

Recall that we have selected a fixed vertex w which serves as the root for all spanning trees constructed during the course of the algorithm. Let T be one of these spanning trees. We define the following *tree indices* for T:

- *predecessor index:* For every vertex $v \neq w$, $p(v)$ is the predecessor of v on the path from w to v in T.
- *depth index:* For every vertex v, $d(v)$ is the distance between v and w in T; thus $d(v)$ is the number of edges on the path from w to v in T. In particular, $d(w) = 0$.

- *thread index:* Let $w, v_1, ..., v_{n-1}$ be an ordering of the vertices according to a depth first search on T with start vertex w. Then we put $th(w) = v_1$, $th(v_{n-1}) = w$, and $th(v_i) = v_{i+1}$ for $i = 1, ..., n-2$. Thus the thread indices are used to describe a possible traversing of the vertex set according to a DFS on T. Note that these indices are, in general, not uniquely determined by T.

In Figure 11.3, we have drawn a spanning tree T with root $w = 0$. Below, the values $p(v)$, $d(v)$, and $th(v)$ are listed (where the DFS on T runs from left to right).



v	0	1	2	3	4	5	6	7	8	9
$p(v)$	–	0	0	1	1	2	2	3	3	5
$d(v)$	0	1	1	2	2	2	2	3	3	3
$th(v)$	1	3	5	7	2	9	0	8	4	6

Fig. 11.3. Tree indices

By adjoining the entering arc e to T, we create the pivot cycle C. With a naive implementation, finding C would require a search of the entire tree T and therefore cause a complexity of $O(|V|)$. Using the predecessor and depth indices, we only need $O(|C|)$ instead. Practical experience shows that this will often result in a speed-up by a factor of about $n/\log n$; thus we may hope to find C about 1000 times faster for $n = 10,000$. Similar speed-ups can be achieved for the potential updates to be discussed later.

The following procedure will determine the apex of C, the value of δ used in cancelling C, and also the leaving arc a as determined by the rule of the last blocking arc.

Algorithm 11.5.1. Let (T, L, U) be the current strongly admissible tree structure, with tree indices p, d, th, entering arc $e = rs$ and pivot cycle C. For $i \neq w$, denote the edge in T joining i and $p(i)$ by $e(i)$, and let $r(i)$ be the amount of flow which may still be sent through $e(i)$ according to the orientation of C.

Procedure PIVOTCYCLE$(G, b, c, d, \gamma, T, L, U, e; \text{apex}, a, \delta)$

```
(1) δ ← ∞;
(2) if e ∈ L then i ← r, j ← s else i ← s, j ← r fi
(3) while i ≠ j do
(4)         if d(i) > d(j)
(5)                 if r(i) < δ then δ ← r(i), a ← e(i) fi
(6)                 i ← p(i)
(7)         else
(8)                 if r(j) ≤ δ then δ ← r(j), a ← e(j) fi
(9)                 j ← p(j)
(10)        fi
(11) od
(12) apex ← i.
```

Note that i and j have been defined in such a way that i is reached before j when C is traversed according to its orientation. The vertices i and j run during the procedure through two disjoint subtrees of T which meet in the apex of C. Thus we have reached the apex in the moment when $i = j$. Then the procedure terminates, and i is the apex of C, the arc a is the last blocking arc of C, and δ is the maximal amount which may be used for augmenting along C according to its orientation (and thus the amount involved in cancelling this cycle).

In order to augment the current tree solution f associated with (T, L, U), it is necessary to traverse C once again. While this is somewhat unfortunate, it does result in an extra benefit which will turn out to be useful in updating the potentials: we can decide if the start vertex r of the entering arc $e = rs$ lies in the subtree T_1 defined in Lemma 11.4.3, via the variable *subtree*.

Algorithm 11.5.2. Let (T, L, U) be the current strongly admissible tree structure, with tree indices p, d, th, entering arc $e = rs$, and pivot cycle C. Moreover, let the apex of C, the value δ involved in cancelling C, and the leaving arc a (as determined by the rule of the last blocking arc) be computed via Algorithm 11.5.1.

Procedure AUGMENT$(G, b, c, d, \gamma, T, L, U, e, \text{apex}, a, \delta; \text{subtree})$

```
(1) i ← r; j ← s;
```

(2) **while** $i \neq$ apex **do**
(3) augment the flow value on $e(i)$ by δ (in the direction of C);
(4) **if** $e(i) = a$ **then** subtree $= T_2$;
(5) $i = p(i)$
(6) **od**
(7) **while** $j \neq$ apex **do**
(8) augment the flow value on $e(j)$ by δ (in the direction of C);
(9) **if** $e(j) = a$ **then** subtree $= T_1$;
(10) $j = p(j)$
(11) **od**

Of course, augmenting the flow value on an edge means either increasing or decreasing it by δ, according to whether the orientation of the edge under consideration agrees or disagrees with that of C.

When we remove the leaving arc $a = uv$ from T, the tree splits into the two connected components T_1 and T_2. By Lemma 11.4.3, the current potential π does not change on T_1 when we switch to the new tree structure by adding the entering arc; and on T_2, the new potential differs from π only by a constant. Note that u is in T_1 if and only if $d(u) < d(v)$. By Lemma 11.4.3, the following procedure correctly updates the potential.

Algorithm 11.5.3. Let (T, L, U) be the current strongly admissible tree structure, with associated potential π and tree indices p, d, th, and let $e = rs$ be the entering arc. Moreover, let the value of subtree be as computed by Algorithm 11.5.2.

Procedure PIUPDATE$(G, b, c, d, \gamma, T, L, U, e, \text{subtree}; \pi)$

(1) **if** subtree $= T_1$ **then** $y \leftarrow v$ **else** $y \leftarrow u$;
(2) **if** $d(u) < d(v)$ **then** $\varepsilon \leftarrow \gamma_\pi(e)$ **else** $\varepsilon \leftarrow -\gamma_\pi(e)$;
(3) $\pi(y) \leftarrow \pi(y) + \varepsilon$, $z \leftarrow th(y)$;
(4) **while** $d(z) > d(y)$ **do** $\pi(z) \leftarrow \pi(z) + \varepsilon$, $z \leftarrow th(y)$ **od**

Of course, there remains the task of efficiently updating the tree indices; as this is quite technical and involved, we will omit a discussion of this topic and refer the reader to [BaJS90] instead.

We have also left another major step in the network simplex algorithm unspecified: the selection of the entering arc in the pricing step (3). Actually the selection strategy chosen for this task plays a decisive role in the overall performance of the algorithm. Practical rules are heuristics: they do not lead to a provably polynomial complexity. An example for a rule that works very well in practice is provided by the *multiple partial pricing* rule employed in the MCFZIB-code mentioned before.

12

Synthesis of Networks

What thought and care to determine the exact site for a bridge, or for a fountain, and to give a mountain road that perfect curve which is at the same time the shortest...

Marguerite Yourcenar

Up to now, we have considered flows or circulations only on a *given* network. But it is also quite interesting to study the reverse problem of *designing* a network (as economically as possible) on which a flow meeting given requirements can be realized. On the one hand, we will consider the case where all edges may be built with the same cost, and where we are looking for an undirected network with lower bounds on the maximal values of a flow between any two vertices. Both the analysis and design of such *symmetric networks* use so-called *equivalent flow trees*; this technique has an interesting application for the construction of certain communication networks which will be the topic of Section 12.4. On the other hand, we shall address the question of how one may increase the maximal value of the flow for a given flow network by increasing the capacities of some edges by the smallest possible amount.

12.1 Symmetric networks

Let $G = (V, E)$ be a graph with a nonnegative *capacity function* $c\colon E \to \mathbb{R}$; we will call $N = (G, c)$ a *symmetric network*. If we want to treat N in the usual manner, we may replace G with its complete orientation $\overrightarrow{G}$ and define c accordingly: then $c(xy) = c(yx)$ for every arc xy. Let us assume that G is connected, so that $\overrightarrow{G}$ is strongly connected. Then any two distinct vertices s and t of G define an ordinary flow network : $N_{st} = (\overrightarrow{G}, c, s, t)$. We will denote the maximal value of a flow on N_{st} by $w(s, t)$. Note that w is a symmetric function: $w(s, t) = w(t, s)$. We call $w : V \times V \to \mathbb{R}$ the *flow function* of the symmetric network N; for the sake of simplicity, we put $w(x, x) = 0$ for $x \in V$. In Section 12.3, we will consider the construction of symmetric networks for a prescribed flow function. In the present section, we study the basic question of which symmetric functions w occur as flow functions. The following theorem due to Gomory and Hu [GoHu61] gives a simple answer.

Theorem 12.1.1. *Let V be a set with n elements, and let $w\colon V \times V \to \mathbb{R}_0^+$ be a symmetric function. Then there exists a symmetric network $N = (G, c)$ with flow function w on an appropriate connected graph $G = (V, E)$ if and only if the following inequality holds whenever x, y, z are three distinct elements of V:*

$$w(x, y) \;\geq\; \min\{w(x, z), w(z, y)\}. \tag{12.1}$$

Proof. First let $N = (G, c)$ be an arbitrary symmetric network on the vertex set V, and let x, y, z be any three distinct elements of V. By Theorem 6.1.6, there exists a cut (S, T) with $x \in S$ and $y \in T$ such that $w(x, y) = c(S, T)$. If z is contained in S, we obtain $w(z, y) \leq c(S, T) = w(x, y)$ by Lemma 6.1.2; and if z is in T, we have $w(x, z) \leq c(S, T) = w(x, y)$. Thus condition (12.1) is satisfied for every flow function w.

Conversely, let w be a symmetric function satisfying (12.1). We consider the complete graph K on V with weight function w and choose a maximal spanning tree T of (K, w), as in Section 4.5. Now let x, y be any two distinct vertices. By Theorem 4.5.4, the unique path from x to y in T is a path of maximal capacity with respect to w; we denote this capacity by $q(x, y)$. Then $q(x, y) \geq w(x, y)$, because the edge xy is also a path from x to y. Using induction, (12.1) implies

$$w(x_1, x_k) \;\geq\; \min\{w(x_1, x_2), \ldots, w(x_{k-1}, x_k)\} \tag{12.2}$$

for each $k \geq 3$ and any k distinct vertices $x_1, \ldots, x_k$. If we choose the vertices on a path of maximal capacity from x to y for $x_1, \ldots, x_k$, then (12.2) implies $w(x, y) \geq q(x, y)$, and hence equality holds. Now put $c(e) = w(u, v)$ for every edge $e = uv$ of T, and choose $G = T$ and $N = (T, c)$. As the path $P_{x,y}$ from x to y in T is uniquely determined, the maximal value of a flow from x to y in N equals the capacity $q(x, y) = w(x, y)$ of $P_{x,y}$. Therefore w is the flow function of the symmetric network (T, c). □

Corollary 12.1.2. *Every flow function on a symmetric network can also be realized on a tree. If the symmetric network N is defined on n vertices, the flow function on N takes at most $n - 1$ distinct values.*

Proof. The first claim follows from the proof of Theorem 12.1.1. The second claim is clear: as a tree on n vertices has exactly $n - 1$ edges, at most $n - 1$ distinct weights and hence at most $n - 1$ distinct capacities occur. □

The following further consequence of Theorem 12.1.1 will be used in the next section.

Corollary 12.1.3. *Let N be a symmetric network with flow function w. Then the smallest two of the three values $w(x, y)$, $w(x, z)$, and $w(y, z)$ coincide whenever x, y, z are three distinct elements of V.*

Proof. We may assume that $w(x, y)$ is the minimum of the three values in question. Condition (12.1) shows that the two inequalities $w(x, y) < w(x, z)$ and $w(x, y) < w(y, z)$ cannot hold simultaneously. □

Exercise 12.1.4. Let $N = (G, c)$ be a symmetric network with flow function w, and assume that G is a complete graph. Show that a spanning tree T of G is an equivalent flow tree for N if and only if T is a maximal spanning tree for the network (G, w). Here a tree T is called an *equivalent flow tree* for N if the flow function of the symmetric network $(T, w|T)$ is equal to w.

Exercise 12.1.5. Show that every flow function may be realized on a path, and give such a realization for the symmetric network of Figure 12.1.

Hint: Consider a pair (x, y) of vertices such that $w(x, y)$ is maximal, and use induction on the number of vertices.

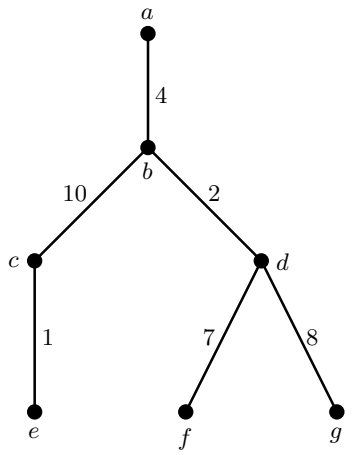


Fig. 12.1. A symmetric network on a tree

Now let $r\colon V \times V \to \mathbb{R}_0^+$ be a symmetric function which describes the flow requirements for which we would like to construct a symmetric network: the *request function*. We will allow the possibility that these flow requirements do not satisfy the necessary condition (12.1) for a flow function, so that it is not possible to realize these requirements exactly. This suggests the following definitions. A network $N = (G, c)$ on a connected graph $G = (V, E)$ on V is called *feasible* for a given request function r if the condition $w(x, y) \geq r(x, y)$ is satisfied all $x, y \in V$. A *minimal network* for r is a feasible network for which the overall capacity

$$c(E) \;=\; \sum_{e \in E} c(e)$$

is minimal among all feasible networks: we want to minimize the sum of the capacities of all the edges we have to provide. This makes sense, for example, if the cost of building an edge depends linearly on the capacity we want to install only, but not on the choice of the edge. Of course, this assumption is

not always realistic: in traffic networks, for example, the cost of building an edge will usually depend both on the terrain and on the distance between the start and the end vertex. Hence a cost function of the form $\sum \gamma(e)c(e)$ would clearly be more useful; it is possible to treat this case as well, albeit with considerably more effort; see [GoHu62]. The special case we consider here admits a particularly elegant solution which we shall present in Section 12.3. Before doing so, let us have a closer look at analyzing symmetric networks and at the synthesis of equivalent flow trees.

12.2 Synthesis of equivalent flow trees

In this section, we provide an efficient way to analyze a given symmetric network $N = (G, c)$ on a graph $G = (V, E)$ – that is, to determine its flow function w. Recall that we may calculate with complexity $O(|V|^3)$ the value of the flow between any two given vertices, by Theorems 6.4.8 and Theorem 6.6.15. As the number of pairs of vertices is $|V|(|V|-1)/2$, the flow function can certainly be determined with a complexity of $O(|V|^5)$: just run one of the standard algorithms for all pairs of vertices. However, Corollary 12.1.2 suggests that we ought to be able to do better, since there are at most $|V|-1$ distinct flow values. This is indeed possible: Gomory and Hu [GoHu61] proved that it suffices to compute $|V|-1$ flow values on suitable (smaller) networks obtained from N by *condensing*, which leads to a complexity of $O(|V|^4)$ for determining w. A detailed description of their rather complicated algorithm can be found in §IV.3 of [FoFu62]. We present an alternative, considerably simpler technique due to Gusfield [Gus90] which likewise works with computing just $|V|-1$ flow values.

As we saw in the proof of Theorem 12.1.1, there exist a spanning tree T on V and a weight function $w\colon T \to \mathbb{R}_0^+$ such that the capacity of the path from x to y is equal to the value $w(x, y)$ of the flow function for all pairs $x, y \in V$. This means that T is an equivalent flow tree for N.[1] We shall present Gusfield's algorithm in a rather concise and informal way. The algorithm requires the determination of a minimal cut (S, T) for a flow network (G, c, s, t). Such a cut may be found by applying a labelling procedure (see Corollary 6.1.4) after a maximal flow has been found. More precisely, this task can be performed with complexity $O(|E|)$ if we apply the procedure AUXNET modified according to Exercise 6.3.19. Therefore we may use any algorithm for determining maximal flows as a subroutine for finding the required cuts in Gusfield's algorithm.

Algorithm 12.2.1. Let $N = (G, c)$ be a symmetric network on $G = (V, E)$, where $V = \{1, \ldots, n\}$. The algorithm determines an equivalent flow tree (B, w). It also calculates a function p; for $i \neq 1$, $p(i)$ is the predecessor of

[1] Using this rather sloppy notation (that is, using the same symbol w for the weight function on T as well as for the flow function on N) is justified, as $w(x, y) = w(e)$ for each edge $e = xy$ of T.

i on a path from vertex 1 to vertex i in B; thus B consists of the edges $\{p(i), i\}$ for $i = 2, \ldots, n$.

Procedure FLOWTREE$(G, c; B, w)$

(1) $B \leftarrow \emptyset$;
(2) **for** $i = 2$ **to** n **do** $p(i) \leftarrow 1$ **od**
(3) **for** $s = 2$ **to** n **do**
(4) $\quad t \leftarrow p(s)$;
(5) $\quad$ calculate a minimal cut (S, T) and the value w of a maximal flow in the flow network (G, c, s, t);
(6) $\quad B \leftarrow B \cup \{st\}$; $w(s, t) \leftarrow w$;
(7) $\quad$ **for** $i = s + 1$ **to** n **do**
(8) $\quad\quad$ **if** $i \in S$ **and** $p(i) = t$ **then** $p(i) \leftarrow s$ **fi**
(9) $\quad$ **od**
(10) **od**

Note that the function p in the procedure FLOWTREE defines a spanning tree B on V throughout the algorithm: B is initialized in step (2) as a star with center 1. During the s-th iteration, B is a tree for which all vertices $i \geq s$ are leaves, where $p(i)$ gives the unique neighbor of i in B. The neighbor of s is chosen as the sink t, and a minimal cut (S, T) for the network with source s and sink t is calculated. Then one assigns the maximal value of a flow from s to t as weight $w = c(S, T)$ to the edge st. Finally, in steps (7) to (9), we cut off all leaves $i > s$ which satisfy $p(i) = t$ and are contained in S, and re-connect all these vertices to s as leaves. Before proving that B is indeed an equivalent flow tree for the given weight function when the algorithm terminates, let us give an example.

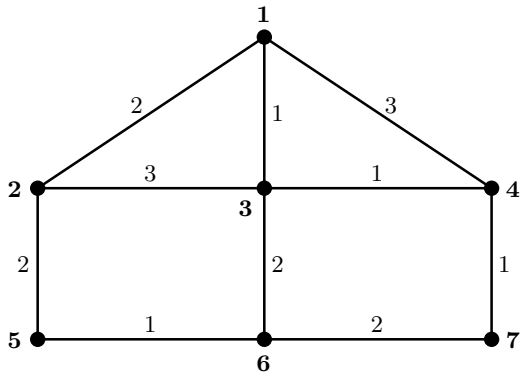


Fig. 12.2. A symmetric network

Example 12.2.2. Consider the network of Figure 12.2, where the edge labels give the capacities. Figure 12.3 shows the star with center 1 constructed during

the initialization and the tree resulting from the iteration for $s = 2$. During this iteration, $t = 1$ and $w(s,t) = 5 = c(S,T)$ for the cut $S = \{2,3,5,6,7\}$ and $T = \{1,4\}$. (In this simple example, it is not necessary to appeal to a max-flow algorithm: the values of the flow and a minimal cut can always be found by inspection.) Now the leaves $3, 5, 6$, and 7 are cut off from $t = 1$ and connected to $s = 2$ instead, and the edge $\{1,2\}$ is assigned weight 5.

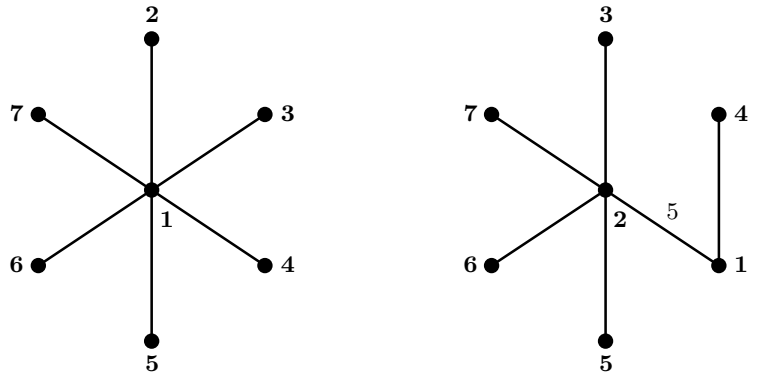


Fig. 12.3. Initialization and iteration $s = 2$

During the next iteration, $s = 3$, $t = 2$, and $w(S,T) = 6 = c(S,T)$ with $S = \{1,3,4,6,7\}$ and $T = \{2,5\}$. The leaves 6 and 7 are cut off from $t = 2$ and connected to $s = 3$; edge $\{2,3\}$ is assigned weight 6. This yields the tree on the left hand side of Figure 12.4. This tree is not changed during the two subsequent iterations, but two more edges are assigned their weights. For $s = 4$, we have $t = 1$ and $w(s,t) = 5 = c(S,T)$ with $S = \{4\}$, $T = \{1,2,3,5,6,7\}$; and for $s = 5$, we have $t = 2$ and $w(s,t) = 3 = c(S,T)$ with $S = \{5\}$ and $T = \{1,2,3,4,6,7\}$.

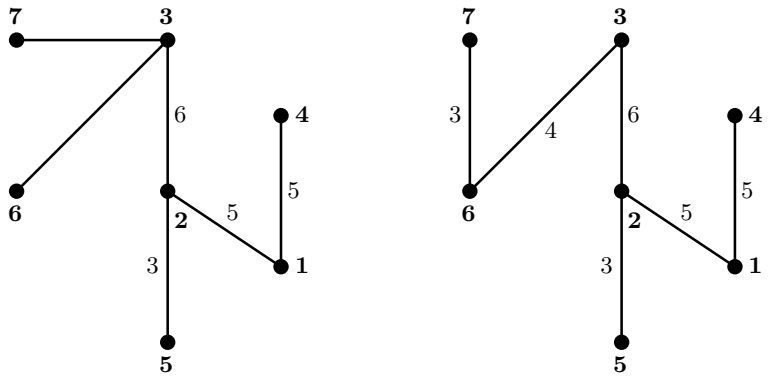


Fig. 12.4. Iterations $s = 3, 4, 5$ and $s = 6, 7$

The next iteration yields a new tree: For $s = 6$ and $t = 3$, we get $w(s,t) = 4 = c(S,T)$ for $S = \{6,7\}$ and $T = \{1,2,3,4,5\}$. Thus vertex 7 is cut off from vertex 3 and connected to vertex 6, yielding the tree on the right hand side of Figure 12.4. This tree remains unchanged during the final iteration: for $s = 7$, we have $t = 6$, $w(s,t) = 3 = c(S,T)$ with $S = \{7\}$ and $T = \{1,2,3,4,5,6\}$. It is easy to check that this final tree is indeed an equivalent flow tree.

To show that Algorithm 12.2.1 is correct, we need some preliminaries. The following lemma comes from [GoHu61].

Lemma 12.2.3. *Let $N = (G,c)$ be a symmetric network and (X,Y) a minimal (x,y)-cut – that is, a minimal cut in the flow network (G,c,x,y) – for two distinct vertices x and y of G. Moreover, let u and v be two vertices in X and (U,V) a minimal (u,v)-cut. If $y \in U$, then $(U \cup Y, V \cap X)$ is also a minimal (u,v)-cut; and if $y \in V$, then $(U \cap X, V \cup Y)$ is a minimal (u,v)-cut.*

Proof. We may assume that none of the four sets

$$P = X \cap U, \quad Q = Y \cap U, \quad R = X \cap V \quad \text{and} \quad S = Y \cap V$$

is empty; otherwise the assertion is trivial. (For example, for $Q = \emptyset$, we have $U \cap X = U$ and $V \cup Y = V$.) Then (X,Y) and (U,V) are said to be *crossing cuts*. Thus our goal is to construct a minimal (u,v)-cut (U',V') such that (X,Y) and (U',V') are *non-crossing cuts*. Figure 12.5 illustrates the given crossing cuts for the two possible cases for y. Using symmetry arguments, it suffices to consider one of these two cases, say $y \in Q$. Note that it does not matter whether $x \in U$ or $x \in V$.

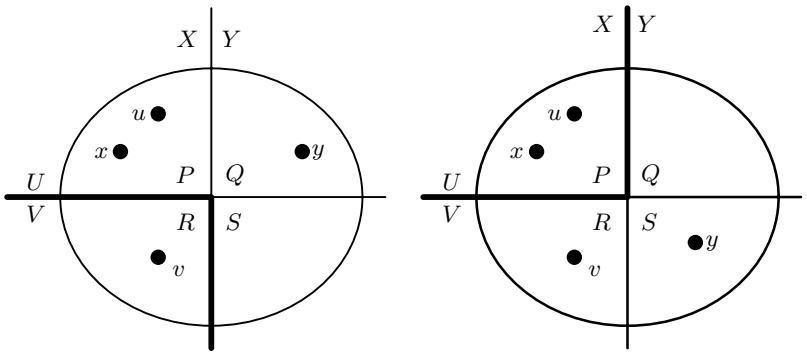


Fig. 12.5. The crossing cuts in Lemma 10.2.3

As $(P \cup R \cup S, Q)$ is an (x,y)-cut and as (X,Y) is a minimal (x,y)-cut, we obtain

$$\begin{aligned} c(P,Q) + c(P,S) + c(R,Q) + c(R,S) &= c(X,Y) \\ &\leq c(P \cup R \cup S, Q) \\ &= c(P,Q) + c(R,Q) + c(S,Q) \end{aligned}$$

and therefore

$$c(P,S) + c(R,S) \;\leq\; c(S,Q).$$

Using the trivial inequality $c(P,S) \geq 0$, we conclude $c(R,S) \leq c(S,Q)$, so that

$$\begin{aligned} c(P \cup Q \cup S, R) \;&=\; c(P,R) + c(Q,R) + c(S,R) \\ &\leq\; c(P,R) + c(Q,R) + c(Q,S) + c(P,S) \;=\; c(U,V), \end{aligned}$$

where we have used the symmetry of c. As (U,V) is a minimal (u,v)-cut, $(P \cup Q \cup S, R) = (U \cup Y, V \cap X)$ has to be a minimal (u,v)-cut as well. □

Corollary 12.2.4. *Under the assumptions of Lemma 12.2.3, there exists a minimal (u,v)-cut (U',V') with $U \cap X = U' \cap X$ for which (X,Y) and (U',V') are non-crossing cuts.* □

We now turn to analyzing the procedure FLOWTREE. In each iteration, we view the edge $\{s,t\}$ treated in step (6) as oriented from s to t. Then the tree B generated by the algorithm is oriented in such a way that $s > t$ holds for each edge st of B. Note that all directed paths in B are oriented towards vertex 1; that is, B has the opposite orientation of a spanning arborescence with root 1. For the tree of Example 12.2.2, the orientation is shown in Figure 12.6.

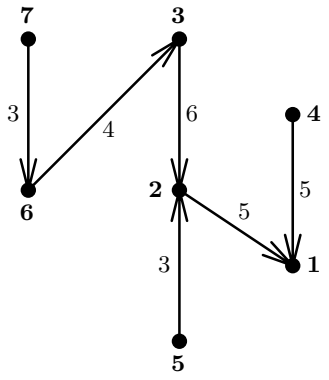


Fig. 12.6. Oriented flow tree for Example 10.2.2

Lemma 12.2.5. *Let B be a directed tree generated by Algorithm 12.2.1 for a symmetric network $N = (G,c)$. Moreover, let $P = P_{ij}$ be a directed path in B with start vertex i and end vertex j, and let kj be an arc in B such that $k \leq h$ holds for each vertex $h \neq j$ on P. Then, at the point of the algorithm when a minimal (k,j)-cut $C = (K,J)$ is constructed, i is adjacent to j in the current tree. Moreover, $i \in K$ if and only if k is a vertex of P.*

Proof. After the initialization phase, each vertex $h \neq 1$ is a leaf, with 1 as its unique neighbor. Note that vertex h stays a leaf until iteration h. The neighbor of h can change (from u to v, say) only if $s = v$ and $t = u$ in some iteration. It is easy to see that the directed path P_{h1} from h to 1 in the tree B consists precisely of those vertices which were the unique neighbor of h at some point during the first h iterations of the algorithm. Now each path P_{ij} is part of the path P_{i1}; hence i, while it was still a leaf, must have been adjacent to j at some point of time before iteration i was executed.

Assume that the neighbor of i was changed afterwards; necessarily, this happened during the iteration $s = h$ (where $t = j$) for the predecessor h of j on P_{ij}. However, $k \leq h$ holds by hypothesis, so that i must have been adjacent to j during the iteration $s = k$, when the (k, j)-cut (K, J) was calculated. Now if k is a vertex on the path P (that is, $k = h$ is the predecessor of j on P), then i must have been contained in K, because otherwise it would not have been cut off from j. Conversely, if $i \in K$, then i is indeed cut off from j and connected to k in step (8) of the algorithm; hence $k \in P$, as asserted. □

Theorem 12.2.6. *Algorithm 12.2.1 determines an equivalent flow tree for the symmetric network N.*

Proof. We introduce some notation first. For any two vertices x and y, let P_{xy} be the unique path from x to y in the tree B determined by Algorithm 12.2.1. Note that, in general, P_{xy} is not a directed path. Moreover, for any path P in B, let $k(P)$ be the capacity of P in the network (B, w): $k(P)$ is the minimum of the values $w(xy)$ over all edges $xy \in P$. Thus the assertion is equivalent to

$$k(P_{xy}) \;=\; w(x, y) \quad \text{for all } x, y \in V, \tag{12.3}$$

where w is the flow function on N. For each edge $xy \in B$ (with $x > y$), let (S_{xy}, T_{xy}) denote the minimal (x, y)-cut which the algorithm calculates in step (5) of the iteration where $s = x$ and $t = y$; we always assume $x \in S_{xy}$ and $y \in T_{xy}$. To prove (12.3), we distinguish four cases. Note that $k(P_{xy}) \leq w(x, y)$ holds by Theorem 12.1.1.

Case 1: xy is an edge of B, so that $x > y$. Then (12.3) holds trivially, because xy has been assigned the value $w(x, y)$ as weight in step (6) of the iteration $s = x$ in this case.

Case 2: P_{xy} is a directed path from x to y; again, this means $x > y$. We use induction on the length l of the path P_{xy}. The induction basis ($l = 1$) was proved in Case 1. Thus assume $l \geq 2$, and let v be the immediate predecessor of y on P_{xy}. Consider the cut (S_{vy}, T_{vy}). By Lemma 12.2.5, $x \in S_{vy}$. Now Lemma 6.1.2 implies $w(x, y) \leq c(S_{vy}, T_{vy}) = w(v, y)$. By the induction hypothesis, $w(x, v) = k(P_{xv})$, so that

$$k(P_{xy}) \;=\; \min\{k(P_{xv}), w(v, y)\} \;=\; \min\{w(x, v), w(v, y)\}.$$

Now if we had $w(x, y) > k(P_{xy})$, Corollary 12.1.3 would imply

$$w(x,y) \;>\; k(P_{xy}) \;=\; w(x,v) \;=\; w(v,y),$$

contradicting $w(x,y) \leq w(v,y)$ above.

Case 3: P_{yx} is a directed path. This case reduces to Case 2 by interchanging the roles of x and y.

Case 4: Neither P_{xy} nor P_{yx} is a directed path. Let z be the first common vertex of the directed paths P_{x1} and P_{y1}. Then P_{xy} is the union of the two directed paths P_{xz} and P_{yz}. Denote the predecessors of z on P_{xz} and P_{yz} by x' and y', respectively. We may assume $x' < y'$, so that the cut $(S_{x'z}, T_{x'z})$ is calculated at an earlier point than the cut $(S_{y'z}, T_{y'z})$. Then the cases treated before imply

$$w(x,z) = k(P_{xz}) \quad \text{and} \quad w(y,z) = k(P_{yz}),$$

so that

$$k(P_{xy}) \;=\; \min\{w(x,z), w(y,z)\}.$$

Now suppose $w(x,y) > k(P_{xy})$. Then Corollary 12.1.3 yields

$$k(P_{xy}) \;=\; w(x,z) \;=\; w(y,z).$$

Therefore P_{xz} contains some edge of weight $k(P_{xy})$; we choose $e = uv$ as the last edge with this weight on the directed path P_{xz}. Applying Lemma 12.2.5 to the path P_{xv}, we get $x \in S_{uv}$. As we assumed $w(x,y) > k(P_{xy})$, we must also have $y \in S_{uv}$, because otherwise

$$w(x,y) \;\leq\; c(S_{uv}, T_{uv}) \;=\; w(u,v) \;=\; k(P_{xy}),$$

a contradiction. Applying Lemma 12.2.5 to the path P_{yz}, we also get $y \notin S_{x'z}$. Hence $uv \neq x'z$, as $y \in S_{uv}$. Again using Lemma 12.2.5 (now applied to the paths P_{xz}, P_{uz}, and P_{vz}), we see that u, v, and x are all contained in $S_{x'z}$. Thus the situation looks as shown in Figure 12.7; the positions of u, v, x, and y in one of the four quarters are uniquely determined, whereas there are two possibilities for x' and z. Depending on whether $z \in Q$ or $z \in S$ holds, either $(R, P \cup Q \cup S)$ or $(P, Q \cup R \cup S)$ is a minimal (u,v)-cut, by Lemma 12.2.3; this yields the two cases of Figure 12.7. We denote this cut by (U,V).

First consider the case $z \in Q$. Then the cut (U,V) separates the vertices z and v, so that

$$w(v,z) \;\leq\; c(U,V) \;=\; c(S_{uv}, T_{uv}) \;=\; w(u,v) \;=\; k(P_{xy}).$$

On the other hand, the fact that the path P_{vz} is directed implies $w(v,z) = k(P_{vz})$. By the choice of e, we must have $k(P_{vz}) > k(P_{xy})$, contradicting the inequality above. Therefore, this case cannot occur and z must be in S. Now the cut (U,V) separates the vertices x and y and we obtain

$$w(x,y) \;\leq\; c(U,V) \;=\; c(S_{uv}, T_{uv}) \;=\; w(u,v) \;=\; k(P_{xy}),$$

that is, $w(x,y) = k(P_{xy})$. □

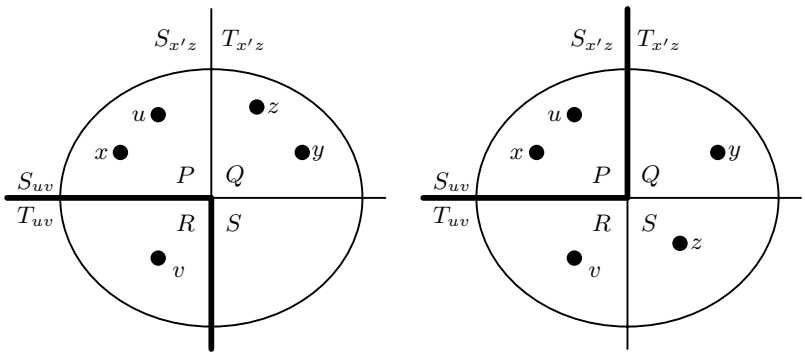


Fig. 12.7. Case 4

Corollary 12.2.7. *Let N be a symmetric network on $G = (V, E)$. Then one may determine with complexity $O(|V|^3|E|^{1/2})$ an equivalent flow tree for N.*

Proof. The assertion follows immediately from Theorems 6.6.17 and 12.2.6, if we use Algorithm 6.6.16 for determining a maximal flow and – as explained at the beginning of this section – a minimal cut in step (5) of procedure FLOWTREE. □

12.3 Synthesizing minimal networks

As promised at the end of Section 12.1, we now present the method of Gomory and Hu [GoHu61] for constructing a minimal feasible network (N, c) for a given request function $r\colon V \times V \to \mathbb{R}$. To this end, we consider the complete graph K on V with weight function r. In the present context, a maximal spanning tree T for (K, r) will be called a *dominant requirement tree* for r. Such a tree T may be determined in $O(|V|^2)$ steps, using the algorithm of Prim (Algorithm 4.4.3) modified for maximal spanning trees as in Section 4.5. Then we partition T into uniform trees, where a graph with a weight function is called *uniform* if all edges have the same weight. To do so, let m be the minimal weight occurring in T, and choose as a first uniform tree the tree T' containing the same edges as T, but each edge with weight m. Now delete all edges of weight $r(e) = m$ from T, and replace the weight $r(e)$ of all other edges by $r(e) - m > 0$. The result is a forest on V; the trees contained in this forest may then be partitioned into uniform trees using the same procedure.

Example 12.3.1. Let K be the graph in Figure 12.8, where edges of weight $r(e) = 0$ are not drawn. The fat edges form a dominant requirement tree T which can be partitioned into uniform trees $U_1, \ldots, U_6$ as in Figure 12.9. Note

that T is not uniquely determined: for instance, the edge gh of T could be replaced by the edge bg.

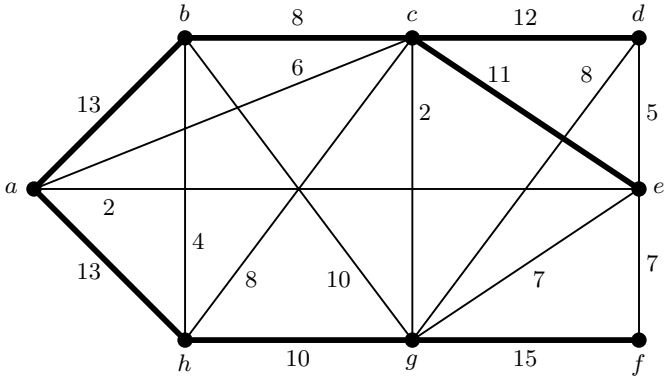


Fig. 12.8. A dominating tree

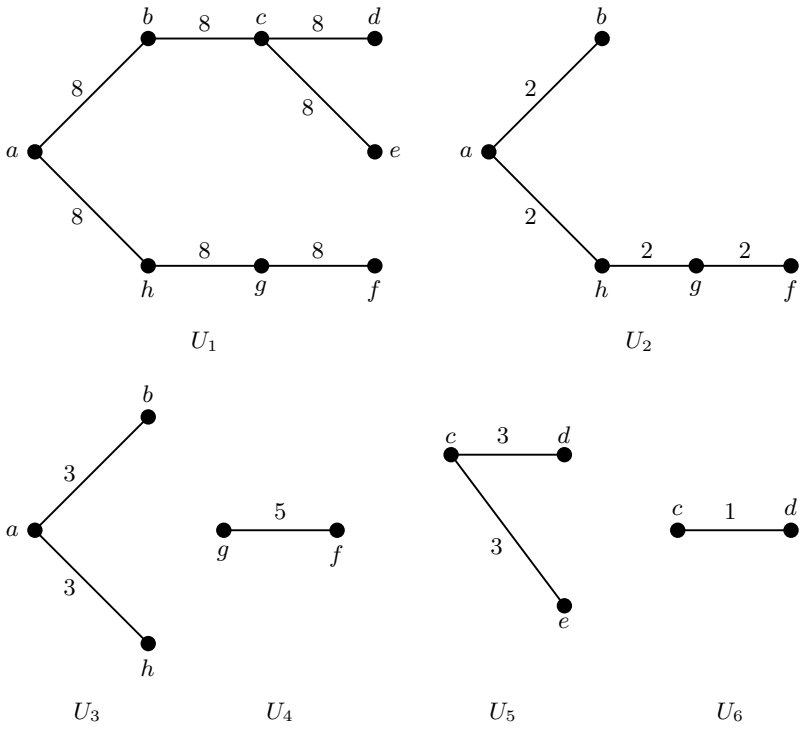


Fig. 12.9. Partitioning T into uniform trees

Suppose that the dominant requirement tree T has been partitioned into uniform trees $U_1, \ldots, U_k$. For each tree U_i containing at least three vertices, we form a cycle C_i on the vertices of U_i, in an arbitrary order; each edge in this cycle is assigned weight $u_i/2$, where u_i is the weight of the edges in U_i. Trees U_i consisting of one edge only are kept as C_i with unchanged weight. Now consider the graph $G = (V, E)$ whose edge set is the union of the edge sets of $C_1, \ldots, C_k$, where parallel edges are merged to form one edge with weight the sum of the individual weights.

Example 12.3.2. For the partition of Figure 12.9, we may use the cycles $C_1, \ldots, C_6$ shown in Figure 12.10 to obtain the symmetric network (G, c) shown in Figure 12.11.

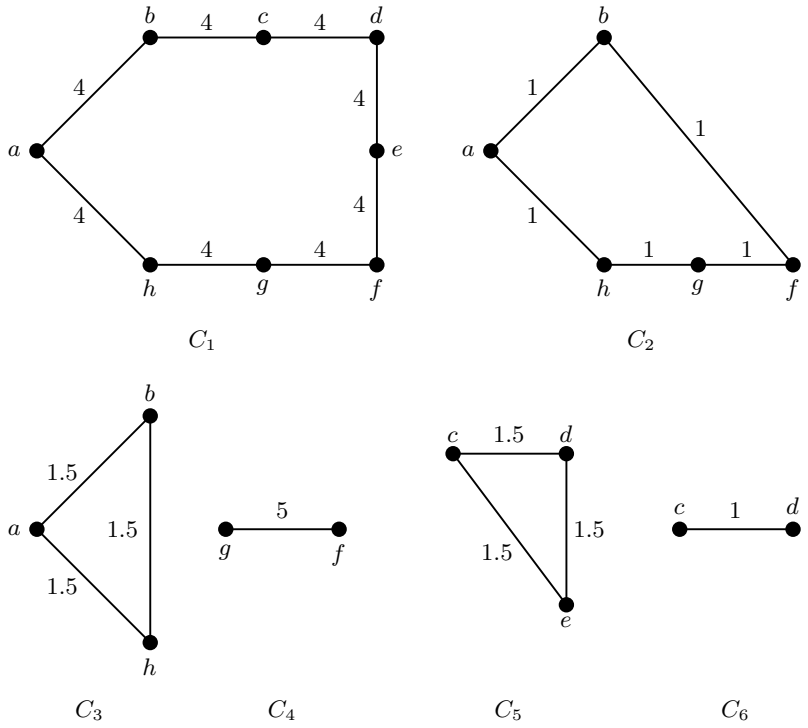


Fig. 12.10. The cycles corresponding to the trees of Figure 12.9

We claim that any symmetric network (G, c) constructed in this way is a minimal feasible network for r. We first prove that N is feasible; it will suffice to verify the following condition:

$$w(u, v) \geq r(u, v) \quad \text{for each edge } uv \text{ of } T. \tag{12.4}$$

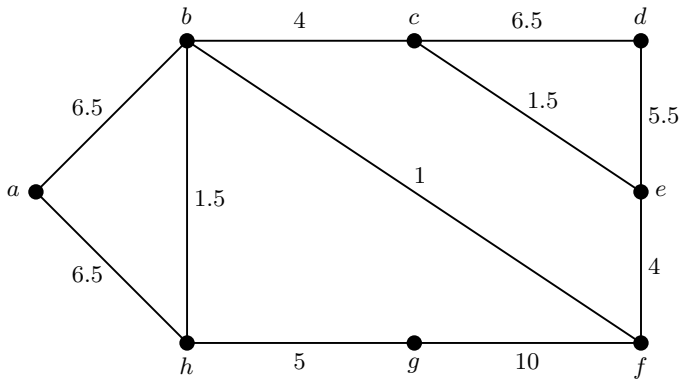


Fig. 12.11. The corresponding symmetric network

Clearly, this condition is necessary. On the other hand, for any two vertices x and y, the unique path P in T from x to y is a path of maximal capacity in K, by Theorem 4.5.4. Now (12.2) in the proof of Theorem 12.1.1 and (12.4) imply

$$w(x,y) \;\geq\; \min\{w(u,v)\colon uv \in P\} \;\geq\; \min\{r(u,v)\colon uv \in P\} \;\geq\; r(x,y),$$

since xy is a path of capacity $r(x,y)$ in K.

Thus we have to check that condition (12.4) is satisfied for the symmetric network (G,c) defined above. But this is rather obvious: for each cycle C_i, we may realize a flow of value u_i between any two vertices of C_i. Now let $e = uv$ be an arbitrary edge of T. By summing the flows from u to v for all C_i which contain both u and v, we obtain a flow from u to v which has the required value

$$\sum_{\substack{i\\ u,v\in U_i}} u_i \;=\; r(u,v).$$

It remains to show that N is a minimal network for r. For each vertex x, we let $u(x)$ denote the maximal value of flow required in x:

$$u(x) \;=\; \max\{r(x,y)\colon\, y \neq x\}\,.$$

As $(x, V\setminus x)$ is a cut (for simplicity, we write x instead of $\{x\}$), Theorem 6.1.6 yields $c'(x, V\setminus x) \geq u(x)$ for every symmetric network $N' = (G', c')$ which is feasible for r. Summing this over all vertices x gives

$$\sum_{x,y\in V} c'(x,y) \;\geq\; \sum_{x\in V} u(x) \;=:\; u(V), \tag{12.5}$$

where $c'(x,y) = 0$ whenever xy is not an edge of G'. Therefore the sum of all capacities in N' is at least $u(V)/2$. We will show that equality holds in (12.5)

for the network N constructed above, so that N is indeed minimal. To this end, we define a function u' on V by

$$u'(x) \;=\; \max\{r(x,y)\colon xy \text{ is an edge of } T\}\,;$$

trivially, $u'(x) \leq u(x)$ for all x.[2] By construction of N, $c(x, V \setminus x) = u'(x)$ holds for every vertex x, so that in N

$$\sum_{x,y\in V} c(x,y) \;=\; \sum_{x\in V} u'(x) \;\leq\; u(V).$$

Thus equality holds in (12.5) for $N' = N$, as claimed.

Finally, let us discuss the complexity of this construction procedure. The dominant requirement tree may be determined $O(|V|^2)$ steps if we use the algorithm of Prim; see Theorem 4.4.4. As there are at most $|V| - 1$ distinct weights, $O(|V|^2)$ steps also suffice to partition T into uniform trees. Finally, constructing the network (G, c) from the uniform trees takes another $O(|V|^2)$ steps. Thus we have proved the following result due to Gomory and Hu:

Theorem 12.3.3. *Let r be a given symmetric request function on a vertex set V. Then one can determine a minimal feasible symmetric network for r in $O(|V|^2)$ steps.* □

We will not write down the preceding algorithm in a formal way; the reader might do so as an exercise. As there are many different choices for the dominant requirement tree T and for the order of the vertices in each of the cycles C_i, there exist many different minimal networks for r. It is possible to single out some of these networks according to a further optimality criterion, although this extra property may still be satisfied by several distinct networks. A minimal network for r will be called *dominating* if its flow function w satisfies the condition

$$w(x,y) \;\geq\; w'(x,y) \qquad \text{for all } x, y \in V,$$

whenever w' is the flow function for some minimal network N' with respect to the request function r. Gomory and Hu also proved that dominating networks indeed exist:

Theorem 12.3.4. *For every request function r on V, there exists a dominating minimal network for r.*

Proof. We replace the given request function r on V by the function s defined as follows:

$$s(x,y) \;=\; \min\{u(x), u(y)\}\,,$$

where u is as before: $u(x) = \max\{r(x,y)\colon y \neq x\}$. The following inequalities show that u can also be defined by using s instead of r:

[2] Actually $u(x) = u'(x)$ for all x, but we do not need this for our proof.

$$\begin{aligned} u(x) &\geq \max\{s(x,y)\colon y \neq x\} = \max\{\min\{u(x),u(y)\}\colon y \neq x\} \\ &\geq \max\{\min\{u(x),r(x,y)\}\colon y \neq x\} \\ &= \max\{r(x,y)\colon y \neq x\} = u(x). \end{aligned}$$

Hence indeed $u(x) = \max\{s(x,y)\colon y \neq x\}$. Now we construct a minimal feasible network N for s. As

$$r(x,y) \leq \min\{u(x),u(y)\} = s(x,y)$$

for all x and y, the network N is also feasible for r. Let us show that in N all flow requirements have to be satisfied with equality: $w(x,y) = s(x,y)$ for all x and y. Suppose otherwise so that $w(x,y) > s(x,y)$ for some $x, y \in V$. We may assume $u(x) \leq u(y)$. Then Lemma 6.1.2 implies

$$c(x, V \setminus x) \geq w(x,y) > s(x,y) = u(x).$$

As N is minimal for s, this contradicts the fact (established in the proof of Theorem 12.3.3) that a minimal network has to satisfy inequality (12.5) with equality. Since the function u is the same for r and s, N has to be a minimal feasible network for r as well.

Finally, let N' be any minimal network for r, with capacity function c' and flow function w'. Moreover, let x and y be any two vertices in V. Suppose $s(x,y) = w(x,y) < w'(x,y)$ and assume w.l.o.g. $s(x,y) = u(x) \leq u(y)$. Applying Lemma 6.1.2 again yields

$$c'(x, V \setminus x) \geq w'(x,y) > w(x,y) = u(x),$$

so that equality cannot hold in (12.5) for N'. This contradicts the minimality of N' and finishes the proof. □

A dominating network is distinguished among all minimal networks for r by the fact that the flow value is as large as possible for each pair of vertices, subject to the condition that the overall cost has to be as small as possible. Any further increase of the value of the flow for even just one pair of vertices would require increasing the sum of the capacities as well, and therefore the cost would increase. We shall discuss this type of problem in Section 12.5.

Exercise 12.3.5. Determine a dominating feasible network N for Example 12.3.1 and check that there are pairs x, y of vertices for which the value of the flow on N is larger than the flow value realized by the minimal network shown in Figure 12.11.

A more general problem of synthesizing a flow network is studied in [GoHu64]. The problem we have discussed is the special case where, at any point of time, there is only a single flow request for just one pair of vertices; this case is called *complete time-sharing* or *multi-terminal network flow*. The other extreme case occurs if all requests r have to be satisfied simultaneously; this leads to *multi-commodity flows*; see [GoMi84] and [FoFu58c]. One may also treat the case where the flow requests are time-dependent; see [GoHu64].

12.4 Cut trees

In this section we consider a strengthening of the notion of equivalent flow trees introduced in Section 12.2 and present an interesting application to the construction of certain communication networks.

Let $N = (G, c)$ be a symmetric network with flow function w, and let B be an equivalent flow tree for N. Moreover, assume that the following condition holds for each pair of vertices x and y:

$(*)$ The cut (U, V) determined by an edge $e = uv$ of minimal weight $w(u, v)$ on the path P_{xy} from x to y in B is a minimal (x, y)-cut.

Then B is called a *cut tree* for N. It is easy to see that it suffices to verify condition $(*)$ for all edges xy of the equivalent flow T. The following example shows that an equivalent flow tree for N is not necessarily a cut tree.

Example 12.4.1. Consider the symmetric network N of Example 12.2.2 and the equivalent flow tree B constructed there – that is, the tree on the right hand side of Figure 12.4. Then the condition for a cut tree is satisfied for all edges of B but one: the only exception is the edge $e = \{2, 3\}$, where the corresponding cut is $S = \{3, 6, 7\}$, $T = \{1, 2, 4, 5\}$ with capacity $c(S, T) = 7$, whereas $w(2, 3) = 6$. However, modifying B slightly yields the cut tree B' for N shown in Figure 12.12.

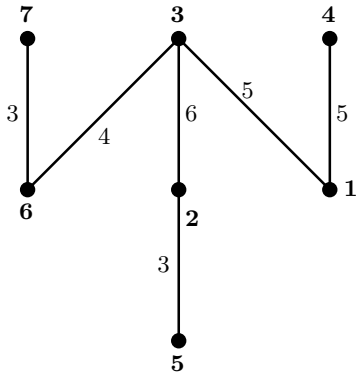


Fig. 12.12. A cut tree

The rather involved method proposed by Gomory and Hu [GoHu61] does in fact always construct not just an equivalent flow tree, but a cut tree. Fortunately, an appropriate modification of the simpler Algorithm 12.2.1 can be used for this purpose as well; the cut tree given in the preceding example was obtained in this way.

Algorithm 12.4.2. Let $N = (G, c)$ be a symmetric network on $G = (V, E)$, where $V = \{1, \ldots, n\}$. The algorithm determines a cut tree (B, w) for N. It also calculates a function p; for $i \neq 1$, $p(i)$ is the predecessor of i on a path from vertex 1 to vertex i in B; thus B consists of the edges $\{p(i), i\}$ for $i = 2, \ldots, n$.

Procedure CUTTREE$(G, c; B, w)$

(1) $B \leftarrow \emptyset$;
(2) **for** $i = 2$ **to** n **do** $p(i) \leftarrow 1$ **od**
(3) **for** $s = 2$ **to** n **do**
(4) $\quad t \leftarrow p(s)$;
(5) $\quad$ determine a minimal cut (S, T) and the value w of a maximal flow in the flow network (G, c, s, t);
(6) $\quad f(s) \leftarrow w$;
(7) $\quad$ **for** $i = 1$ **to** n **do**
(8) $\quad\quad$ **if** $i \in S \setminus \{s\}$ **and** $p(i) = t$ **then** $p(i) \leftarrow s$ **fi**
(9) $\quad$ **od**
(10) $\quad$ **if** $p(t) \in S$ **then** $p(s) \leftarrow p(t)$; $p(t) \leftarrow s$; $f(s) \leftarrow f(t)$; $f(t) \leftarrow w$ **fi**
(11) **od**
(12) **for** $i = 2$ **to** n **do**
(13) $\quad w(i, p(i)) \leftarrow f(i)$;
(14) $\quad B \leftarrow B \cup \{\{i, p(i)\}\}$
(15) **od**

The main difference between Algorithms 12.2.1 and 12.4.2 is as follows: during iteration s of Algorithm 12.2.1, only vertices in S satisfying $i > s$ are cut off from t and re-connected to s; here, in 12.4.2, this is done for vertices $i < s$ as well. When $i < s$, the weight of an edge it which was removed has to be transferred to the new edge is. Moreover, the tree B does not grow by adding edges one by one (as in Algorithm 12.2.1, step (6)), but it might happen that edges $\{s, p(s)\}$ previously constructed are changed again. Thus the procedure CUTTREE is somewhat more involved, which makes the proof of the following result more difficult; in view of its length and technical complexity, we refer the reader to the original paper [Gus90].

Theorem 12.4.3. *Let $N = (G, c)$ be a symmetric network. Then Algorithm 12.4.2 constructs a cut tree for N. By using Algorithm 6.6.16 in step (5), one may achieve a complexity of $O(|V|^3|E|^{1/2})$.* □

A further algorithm for constructing $|V| - 1$ cuts corresponding to a cut tree can be found in [ChHu92]; this algorithm is capable of dealing with an arbitrary symmetric cost function for constructing the cuts – not necessarily the capacity. Related problems are considered in [GuNa91].

Next we use cut trees for treating Problem 4.7.9 (optimum communication spanning tree). As already mentioned in Section 4.7, this problem is NP-complete. However, Hu [Hu74] was able to give an efficient solution for the

special case where all edge weights are equal. Let us formulate this special case explicitly.

Problem 12.4.4 (optimum requirement spanning tree). Let G be a complete graph on the vertex set V, and let $r : V \times V \to \mathbb{R}_0^+$ be a request function. We look for a spanning tree T for G such that

$$\gamma(T) \;=\; \sum_{\substack{u,v \in V \\ u \neq v}} d(u,v) r(u,v)$$

is minimal, where $d(u,v)$ denotes the distance of u and v in the tree T.

Hu's method for solving Problem 12.4.4 rests on finding a cut tree for an appropriate symmetric network. For this purpose, the pair $N = (G,r)$ of Problem 12.4.4 is considered as a symmetric network on G with capacity function r. We begin with the following auxiliary result.

Lemma 12.4.5. *Let G be the complete graph on the vertex set V, and let $r : V \times V \to \mathbb{R}_0^+$ be a request function. Consider the symmetric network $N = (G,r)$. Then every spanning tree T of G satisfies*

$$\gamma(T) \;=\; \sum_{e \in T} r(S_T(e)),$$

where $S_T(e)$ is the cut of G determined by $e \in T$ as in Section 4.3, and where $r(S_T(e))$ denotes the capacity of this cut in N.

Proof. The cost $\gamma(T)$ may be written as follows:

$$\gamma(T) \;=\; \sum_{\substack{u,v \in V \\ u \neq v}} \sum_{e \in P_{uv}} r(u,v) \;=\; \sum_{e \in T} \sum_{\substack{u,v \in V \\ u \neq v,\, e \in P_{uv}}} r(u,v),$$

where P_{uv} denotes the path from u to v in T. Therefore it suffices to show

$$\sum_{\substack{u,v \in V \\ u \neq v,\, e \in P_{uv}}} r(u,v) \;=\; r(S_T(e)).$$

But this is rather obvious: the path P_{uv} in T contains the edge e if and only if u and v lie in different components of the cut $S_T(e)$. □

We need the preceding lemma to establish the following result due to Hu; our proof is considerably simpler than the one in the original paper [Hu74].

Theorem 12.4.6. *Let G be the complete graph on the vertex set V, and let $r : V \times V \to \mathbb{R}_0^+$ be a request function. Then every cut tree T for the symmetric network $N = (G,r)$ is a solution of Problem 12.4.4.*

Proof. Let w be the flow function on N. As we saw in Exercise 12.1.4, every maximal spanning tree B for (G, w) is an equivalent flow tree for N. Let us denote the common weight of all these trees by β; we begin by showing the following auxiliary claim:

$$\gamma(T) \geq \beta \quad \text{for every spanning tree } T \text{ of } G. \tag{12.6}$$

For this purpose, we consider the weight function w' on T defined by

$$w'(u, v) = r(S_T(e)) \quad \text{for all } e = uv \in T.$$

We extend w' to the flow function of the symmetric network (T, w'); using Exercise 12.1.4 again, T is a maximal spanning tree for the network (G, w'). Now (12.6) follows if we can show

$$w(x, y) \leq w'(x, y) \quad \text{for all } x, y \in V. \tag{12.7}$$

Thus let x and y be any two vertices, and choose an edge $e = uv$ of minimal weight with respect to w' on the path P_{xy} from x to y in T. Then indeed

$$w'(x, y) = w'(u, v) = r(S_T(e)) \geq w(x, y),$$

as $S_T(e)$ is an (x, y)-cut. This establishes (12.6), which allows us to restrict our attention to equivalent flow trees for N. Let B be such a tree. Then

$$\gamma(B) = \sum_{e \in B} r(S_B(e)) \geq \sum_{e=uv \in B} w(u, v) = w(B) = \beta,$$

since $S_B(u, v)$ is a (u, v)-cut. Here equality holds if and only if $S_B(u, v)$ is a minimal (u, v)-cut for all $uv \in B$, which means that B is a cut tree for N. □

Theorems 12.4.3 and 12.4.6 immediately yield the following result.

Corollary 12.4.7. *Algorithm 12.4.2 solves Problem 12.4.4 with complexity* $O(|V|^3|E|^{1/2})$. □

Example 12.4.8. Let us interpret the capacity function of the symmetric network N on $V = \{1, \ldots, 7\}$ shown in Figure 12.8 as a request function for the complete graph K_V; of course, we put $r(u, v) = 0$ for edges which are not contained in N. Then the cut tree T displayed in Figure 12.12 solves Problem 12.4.4 for this request function. The weights given in that figure are the capacities of the cuts $S_T(e)$, so that

$$\gamma(T) = \sum_{e \in T} r(S_T(e)) = 26.$$

For comparison, the equivalent flow tree B shown in Figure 12.4 (which was constructed using the simpler Algorithm 12.2.1) has cost

$$\gamma(B) = \sum_{e \in B} r(S_B(e)) = 27,$$

so that B is indeed not an optimal solution for Problem 12.4.4.

We conclude this section with an exercise taken from [Hu74].

Exercise 12.4.9. Determine an optimal solution for the following instance of Problem 12.4.4: $V = \{1, \ldots, 6\}$ and $r(1,2) = 10$, $r(1,6) = 8$, $r(2,3) = 4$, $r(2,5) = 2$, $r(2,6) = 3$, $r(3,4) = 5$, $r(3,5) = 4$, $r(3,6) = 2$, $r(4,5) = 7$, $r(4,6) = 2$, $r(5,6) = 3$.

12.5 Increasing the capacities

The final section of this chapter is devoted to the question of how one may increase the maximal value of a flow on a flow network $N = (G, c, s, t)$ as economically as possible. More specifically, we ask how the capacities of the given edges should be increased if we want to increase the maximal value w of a flow on N by k units; note that we do not allow adding new edges. We shall assume that the cost for increasing the capacity of an edge e by $d(e)$ units is proportional to $d(e)$.

Thus let $N = (G, c, s, t)$ be a flow network with an integral capacity function c, and let $\delta\colon E \to \mathbb{N}$ be an arbitrary mapping. For each $v \in \mathbb{N}$, we look for a mapping $d = d_v\colon E \to \mathbb{N}_0$ for which the network $(G, c + d, s, t)$ allows a flow of value v; moreover, the sum

$$z(v) \;=\; \sum_{e \in E} d(e)\delta(e)$$

should be minimal under this condition. Thus the cost for realizing a flow of value v on N in the prescribed manner is at least $z(v)$, and the function d_v specifies how the capacities should be increased in an optimal solution.

Note that this approach also solves the the *parametric budget problem*, where we want to determine the maximal possible flow value which may be achieved by installing extra capacity subject to a given budget b: we simply need to find the largest value of v for which $z(v) \leq b$ holds. In general, the bound b will not be achieved with equality, as we assumed the capacities to be integral; of course, equality may be obtained by interpolation. The parametric budget problem was first considered by Fulkerson [Ful59]. As we will see, it may be solved using the algorithm of Busacker and Gowen presented in Section 10.5; this is notably simpler than Fulkerson's method.

Let us define a further flow network on a digraph H which extends the given digraph G as follows. Each edge e of G is also contained in H with cost $\gamma(e) = 0$ and with its original capacity: $c'(e) = c(e)$. Additionally, H also contains a parallel edge $e' = uv$ with unlimited capacity $c'(e') = \infty$ and cost$\gamma(e') = \delta(e)$.[3] We claim that our problem can be solved by determining an optimal flow f of value v on the flow network $N' = (H, c', s, t)$, with respect

[3] If desired, we may avoid parallel edges by subdividing e'.

to the cost function γ. This is seen as follows. Since f is optimal, $f(e') \neq 0$ can only hold if e is saturated: $f(e) = c'(e)$. Then the desired function d_v is given by $d_v(e) = f(e')$, and $z(v) = \gamma(f)$.

Hence we may calculate d_v and $z(v)$ for every positive integer v using the algorithm of Busacker and Gowen. Clearly, $z(v) = 0$ for $v \leq w$: in this case, we do not have to increase any capacities. Hence we may start Algorithm 10.5.2 with a maximal flow f on N instead of the zero flow, a further simplification. Let us look at an example.

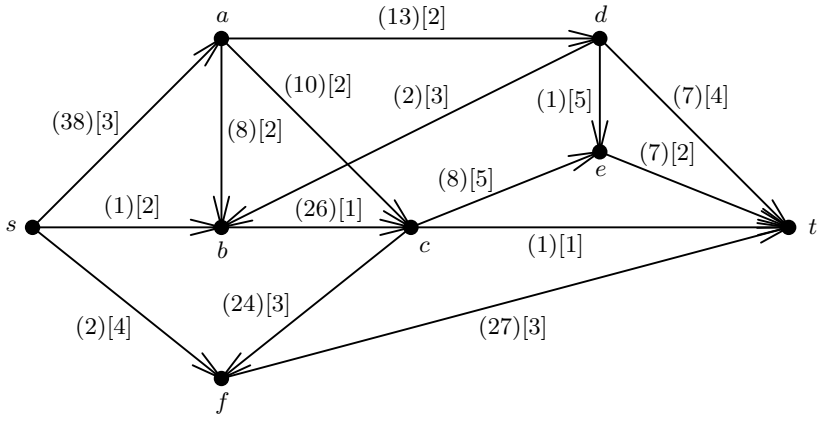


Fig. 12.13. Flow network with cost for capacity increase

Example 12.5.1. Consider again the flow network N of Example 6.2.3; see Figure 12.13, where we state the capacities in parentheses and the cost in brackets. In 6.2.3, we calculated a maximal flow f of value $w = 31$; for the convenience of the reader, we display this flow again in Figure 12.14, where we have also drawn the minimal cut (S, T) corresponding to f. As explained above, we may use f as the initial flow in the algorithm of Busacker and Gowen.

We now have to imagine G as being extended to H: for each edge e of G, we have to add a parallel edge e' with capacity ∞ and cost $\delta(e)$. Then we should proceed by constructing the auxiliary network N^* corresponding to f. Unfortunately, the backward edges and parallel edges make this network rather large and difficult to draw. Therefore we will omit all edges which are not important for our purposes, since they cannot occur in a path of minimal cost from s to t in N^*:

- edges with end vertex s or start vertex t;
- edges e' for which e is not yet saturated;
- edges leading from T to S;[4]

[4] Vertices in S can be reached from s by a path of cost 0, so that we want to direct our path from S to T, not reversely.

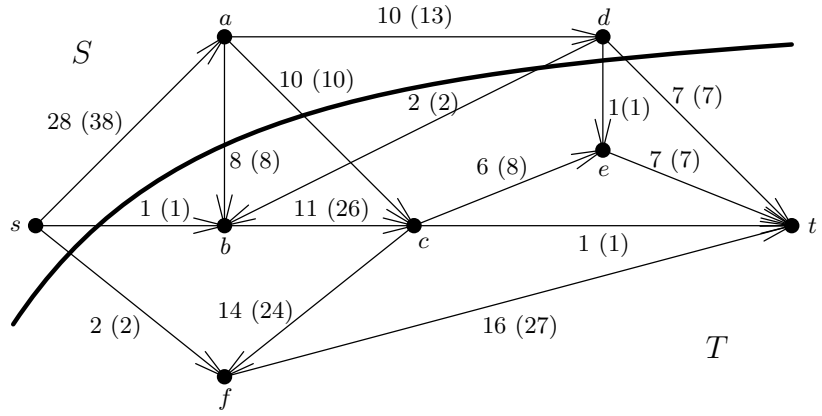


Fig. 12.14. Maximal flow and minimal cut on N

- edges e which are saturated.

The interesting part of N^* is shown in Figure 12.15, where the path P consisting of fat edges is a path of minimal cost from s to t. The numbers without parentheses give the cost, and the numbers in parentheses state the capacities in N^*.

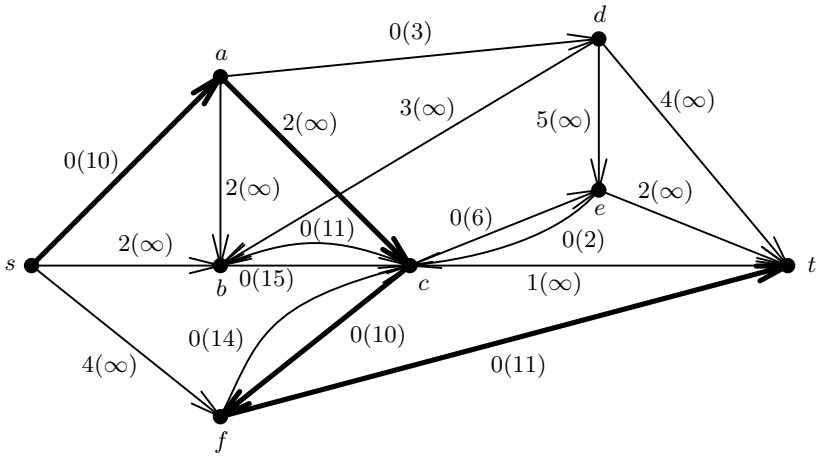


Fig. 12.15. A path of minimal cost in N^*

Note that the path P has cost 2 and capacity 10. Thus we may increase the existing maximal flow of value $w = 31$ by ε with cost 2ε to a flow of value $v = 31 + \varepsilon$ for $\varepsilon = 1, \ldots, 10$. The flow g of value 41 obtained for $\varepsilon = 10$ is shown in Figure 12.16; the fat edge ac is the edge whose capacity was increased. Should we desire any further increase of the value of the flow, we

just continue in the same manner with the algorithm of Busacker and Gowen. If a budget b is given, the procedure may be terminated as soon as we reach $z(v) > b$.

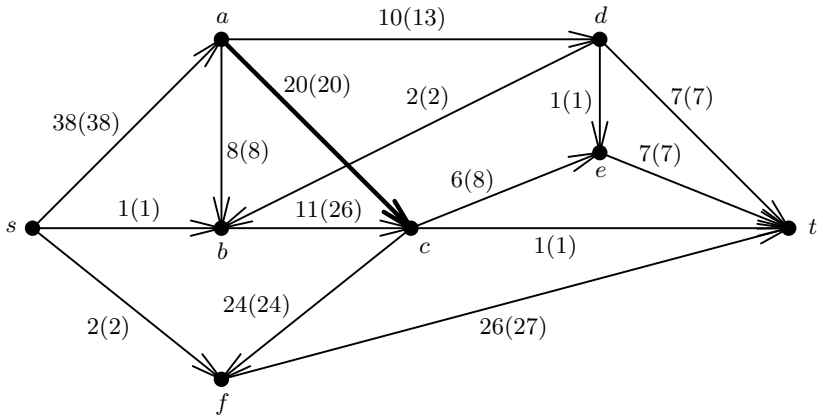


Fig. 12.16. An optimal flow of value 41

Exercise 12.5.2. Determine the cost function $z(v)$ for all v for the flow network of Example 12.5.1. Hint: Two more steps of Algorithm 10.5.2 are needed.

In view of Exercise 10.5.4, it is clear that the cost function $z(v)$ is always a piecewise linear, monotonically increasing, convex function. Moreover, we need at most $|E|$ iterations of the algorithm of Busacker and Gowen to determine $z(v)$ for all v: in the worst case, we have to adjoin a parallel edge for each edge of E to ensure that we reach a path of minimal cost from s to t which has infinite capacity.

A survey of various problems and algorithms concerning the design of networks for communication or transport purposes can be found in [MaWo94].

13

Matchings

And many a lovely flower and tree
Stretch'd their blossoms out to me.
William Blake

This chapter is devoted to the problem of finding maximal matchings in arbitrary graphs; the bipartite case was treated in Section 7.2. In contrast to the bipartite case, it is not at all easy to reduce the general case to a flow problem.[1] However, we will see that the notion of an augmenting path can be modified appropriately.

We emphasize again that the term *maximal matching* means a matching of maximal cardinality, and that an unextendable matching is not necessarily maximal; see Section 7.2. The graphs for which every unextendable matching is already maximal are characterized in [Men74]. An algorithmically satisfactory solution is due to [LePP84]: one can check in polynomial time whether a graph is equimatchable.

13.1 The 1-factor theorem

Clearly, the cardinality of a matching in a graph $G = (V, E)$ cannot exceed $|V|/2$, and a matching of this size exists if and only if $|V|$ is even and G has a 1-factor. As in the bipartite case, a 1-factor is also called a *perfect matching* of G. In this section, we shall prove the 1-factor theorem of Tutte [Tut47] which characterizes the graphs having a perfect matching. This result is a generalization of Hall's marriage theorem (Theorem 7.3.1).

[1] Kocay and Stone [KoSt93, KoSt95] showed that matchings may indeed be treated in the context of flow theory by introducing special types of networks and flows which satisfy certain symmetry conditions: *balanced networks* and *balanced flows*; related ideas can be found in the pioneering work of Tutte [Tut67] and in [GoKa96]. Subsequently, Fremuth-Paeger and Jungnickel provided a general theory based on this approach, including efficient algorithms; see [FrJu99a, FrJu99b, FrJu99c, FrJu01a, FrJu01b, FrJu01c, FrJu02, FrJu03]. We will not present this rather involved theory because that would take up far too much space; instead, we refer the reader to the original papers.

Let $G = (V, E)$ be a graph, and let S be a subset of V. We denote by $p(S)$ the number of connected components of odd cardinality in the graph $G \setminus S$. If G has a perfect matching M, then at least one vertex of each *odd component* (that is, each connected component of odd cardinality) has to be incident with an edge of M whose other vertex is contained in S; hence $p(S) \leq |S|$. The 1-factor theorem states that this necessary condition is actually sufficient. We shall give a short proof taken from [And71].

Theorem 13.1.1 (1-factor theorem). *Let $G = (V, E)$ be a graph. Then G has a perfect matching if and only if the following condition holds:*

(T) $p(S) \leq |S|$ *for each subset S of V,*

where $p(S)$ denotes the number of odd components of $G \setminus S$.

Proof. We have already seen that condition (T) is necessary. Conversely, suppose that this condition holds. For $S = \emptyset$, (T) shows that $|V|$ is even, say $|V| = 2n$. We use induction on n; the case $n = 1$ is trivial. Clearly,

$$p(S) \equiv |S| \pmod 2 \quad \text{for each } S \subseteq V. \tag{13.1}$$

We now distinguish two cases.

Case 1: $p(S) < |S|$ for all subsets S with $2 \leq |S| \leq 2n$. In view of (13.1), actually even $p(S) \leq |S| - 2$. We choose some edge $e = uv$ of G and consider the graph $H = G \setminus A$, where $A = \{u, v\}$. For each subset S of $V \setminus A$, let $p'(S)$ be the number of odd components of $H \setminus S$. Assume that such a set S satisfies $p'(S) > |S|$. Then

$$p(S \cup A) = p'(S) > |S| = |S \cup A| - 2,$$

and hence $p(S \cup A) \geq |S \cup A|$ by (13.1), a contradiction. Hence always $p'(S) \leq |S|$, so that H satisfies condition (T). By the induction hypothesis, H admits a perfect matching M. Then $M \cup \{e\}$ is a perfect matching for G.

Case 2: There exist subsets S of V with $p(S) = |S| \geq 2$. Let us choose a maximal subset R with this property. Then each component of $G \setminus R$ has to be odd. Suppose otherwise, and let C be an even component. Then we can add a vertex a of C to R to obtain a further odd component, which contradicts the maximality of R.

Now let R' be the set of all (necessarily odd) components of $H = G \setminus R$, and consider the bipartite graph B with vertex set $R \dot{\cup} R'$ for which a vertex r in R and a component C in R' are adjacent if and only if there is an edge rc in G with $c \in C$. We show that B has a complete matching, by verifying condition (H) of Theorem 7.2.5. Let J be a set of (necessarily odd) components of H, and let T be the set of vertices in R which are adjacent to some component in J. Now condition (T) for G implies $|J| \leq p(T) \leq |T|$, so that (H) indeed holds for B.

Thus we may choose a vertex x from each component and associate with it a vertex $y_x \in R$ in such a way that xy_x always is an edge in G and all the y_x are distinct. This yields a matching M of G. It now suffices to show that, for each component C (with vertex $x = x(C)$ chosen above), the induced graph G_C on the vertex set $C \setminus x$ has a perfect matching M_C. Then the union of M and all these matchings M_C will yield the desired perfect matching of G.

To this end, we verify condition (T) for G_C. For a subset W of $C \setminus x$, let $p_C(W)$ be the number of odd components of $G_C \setminus W$. Assume $p_C(W) > |W|$ for such a subset W. Then (13.1) implies $p_C(W) \geq |W| + 2$, so that

$$p(W \cup R \cup \{x\}) \;=\; p_C(W) + p(R) - 1 \;\geq\; |W| + |R| + 1 \;=\; |W \cup R \cup \{x\}|.$$

However, this contradicts the maximality of R. Therefore (T) indeed holds for G_C, and our proof is finished. □

Exercise 13.1.2. Let $G = (S \,\dot\cup\, T, E)$ be a bipartite graph with $|S| = |T|$. Show that condition (T) for the existence of a perfect matching in G reduces to condition (H) of Theorem 7.2.5 in this case. Hint: Add the edges of the complete graph on T to G, and consider the resulting graph H instead of G.

Exercise 13.1.3. Let G be a 3-regular graph without bridges. Show that G has a perfect matching [Pet91]. Does this also hold for 3-regular graphs containing bridges? Does a 3-regular graph without bridges necessarily have a 1-factorization?

Next we present a deficiency version of Theorem 13.1.1 – in analogy with the deficiency version of the marriage theorem in Section 7.3. Let M be a matching in the graph $G = (V, E)$. Then a vertex v which is not incident with any edge in M is called *exposed* (with respect to M), whereas vertices incident with some edge of M are called *saturated*. For each saturated vertex v, we will call the unique vertex u with $uv \in M$ the *mate* of v. The following result is due to Berge [Ber58].

Theorem 13.1.4. *Let $G = (V, E)$ be a graph. Then G admits a matching with precisely d exposed vertices if and only if the following condition holds:*

$$d \equiv |V| \pmod 2 \quad \textit{and} \quad p(S) \leq |S| + d \textit{ for all } S \subset V. \tag{13.2}$$

Proof. We define an auxiliary graph H as follows. Adjoin a d-element set D with $D \cap V = \emptyset$ to the vertex set V of G, and add all edges of the form vw with $v \in V$ and $w \in D$ to E. It is now easy to see that G has a matching with precisely d exposed vertices if and only if H has a perfect matching. Thus we have to show that condition (13.2) is equivalent to the existence of a perfect matching of H. For each subset X of $V \cup D$, let $p'(X)$ denote the number of odd components of $H \setminus X$. Obviously, $p'(S \cup D) = p(S)$ for all $S \subset V$. If H has a perfect matching, then condition (T) for H implies immediately

$$p(S) \;=\; p'(S \cup D) \;\leq\; |S \cup D| \;=\; |S| + d \quad \text{for all } S \subset V.$$

Moreover, if H has a perfect matching, $|V| + d$ has to be even, so that (13.2) is necessary.

Conversely, suppose that (13.2) is satisfied. By Theorem 13.1.1, we have to show that the following condition holds:

$$p'(X) \leq |X| \quad \text{for all } X \subset V \cup D. \tag{13.3}$$

Assume first that D is not contained in X. Then $H \setminus X$ is connected by the construction of H, so that (13.3) is obviously satisfied for $X \neq \emptyset$. For $X = \emptyset$, (13.3) holds as $|V \cup D| = |V| + d$ is an even number by hypothesis. Now assume $D \subset X$, say $X = S \dot{\cup} D$ for some $S \subset V$. Then (13.2) implies

$$p'(X) = p(S) \leq |S| + d = |X|,$$

so that (13.3) is satisfied for this case as well. □

Corollary 13.1.5. *Let $G = (V, E)$ be a graph, and let M be a maximal matching of G. Then there are precisely*

$$d = \max\{p(S) - |S| \colon S \subset V\}$$

exposed vertices, and M contains precisely $(|V| - d)/2$ edges.

Proof. The assertion follows immediately from Theorem 13.1.4 together with $|V| \equiv p(S) + |S| \equiv p(S) - |S| \pmod 2$. □

13.2 Augmenting paths

In this section we use *augmenting paths* with respect to a given (not yet maximal) matching M in a graph $G = (V, E)$ for constructing a matching M' of larger cardinality. Let us first consider the bipartite case again. In the solution of Exercise 6.5.5, we determined a maximal matching of G by using a maximal flow on an appropriate 0-1-network. We now want to describe the augmenting paths occurring during this process *within* G. Thus let M be a matching of cardinality k in G, and by f denote the corresponding 0-1-flow (as in the solution of Exercise 6.5.5). Then an augmenting path looks as follows:

$$s \longrightarrow v_1 \longrightarrow v_2 \longleftarrow \ldots \longrightarrow v_{2n-2} \longleftarrow v_{2n-1} \longrightarrow v_{2n} \longrightarrow t,$$

where v_1 and v_{2n} are vertices which are not incident with any saturated edge and where the edges $v_{2i}v_{2i+1}$ are backward edges (that is, they are saturated). Thus the vertices v_1 and v_{2n} are exposed with respect to M, and the edges $v_{2i}v_{2i+1}$ are contained in M; see Figure 13.1, where fat edges are contained in the matching M.

The bipartite case suggests the following way of defining augmenting paths in general. Let M be a matching in an arbitrary graph $G = (V, E)$. An *alternating path* with respect to M is a path P for which edges contained in M alternate with edges not contained in M. Such a path is called an *augmenting path* if its start and end vertex are distinct exposed vertices.

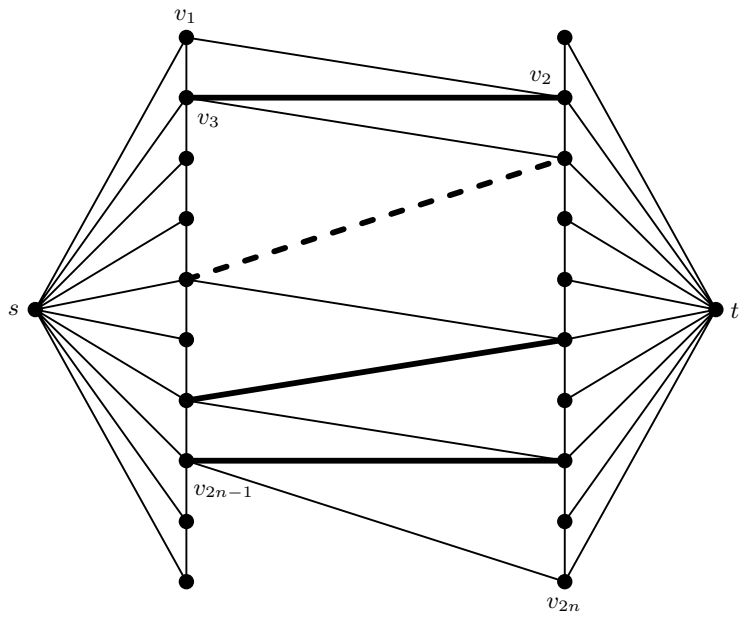


Fig. 13.1. An augmenting path

Example 13.2.1. The fat edges in the graph G displayed in Figure 13.2 form a matching M. The vertices a, f, and y are exposed with respect to M, and the sequences (a, b, c, d, e, f) and (a, b, c, u, v, w, x, y) define augmenting paths P and P', respectively. Interchanging the roles of edges and non-edges of M on the path P' yields the matching M' of cardinality $|M| + 1$ exhibited in Figure 13.3; more formally, we replace M by $M \oplus P'$, where $\oplus$ denotes the symmetric difference. Note that M' is a maximal matching of G, as there is only one exposed vertex.

Example 13.2.1 illustrates the following simple but fundamental result due to Berge [Ber57].

Theorem 13.2.2 (augmenting path theorem). *A matching M in a graph G is maximal if and only if there is no augmenting path with respect to M.*

Proof. Assume first that M is maximal. If there exists an augmenting path P in G, we may replace M by $M' = M \oplus P$, as in Example 13.2.1. Then M' is a matching of cardinality $|M| + 1$, a contradiction.

Conversely, suppose that M is not maximal; we will show the existence of an augmenting path with respect to M. Let M' be any maximal matching, and consider the subgraph H of G determined by the edges in $M \oplus M'$. Note that each vertex of H has degree at most 2; also, a vertex v having degree 2 has to be incident with precisely one edge of M and one edge of M'. Therefore

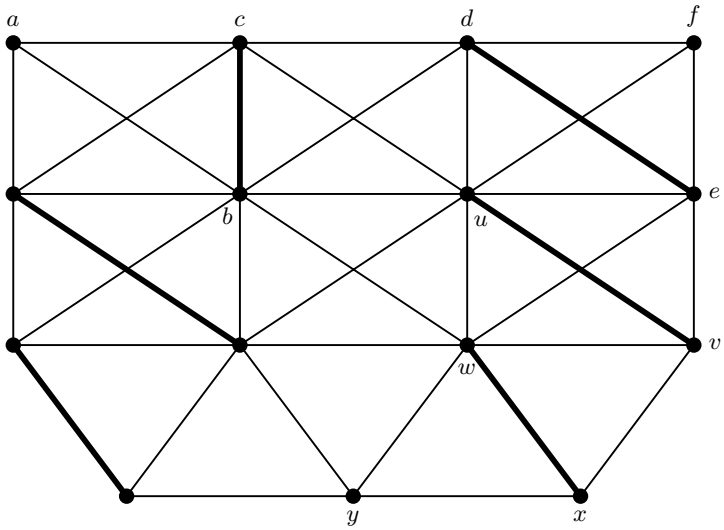


Fig. 13.2. Graph G with matching M

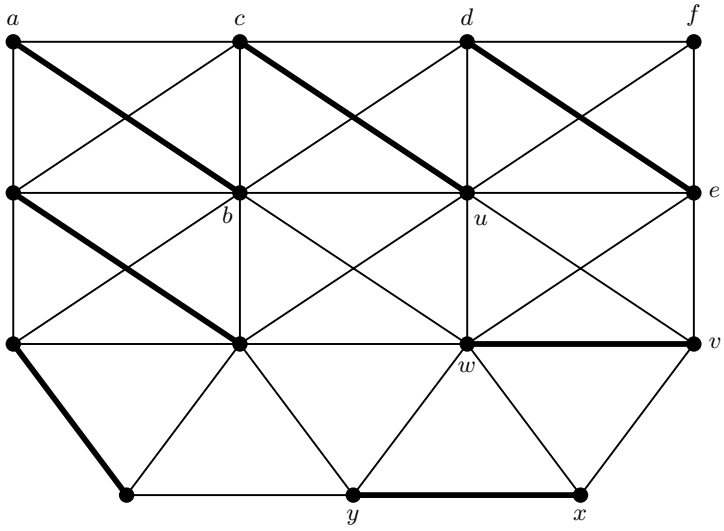


Fig. 13.3. Matching $M' = M \oplus P'$

a connected component of H which consists of more than one vertex has to be either a cycle of even length (where edges of M and M' alternate) or a path formed by such an alternating sequence of edges. As M' contains more edges than M, there exists at least one such path P whose first and last edges belong to M'. Then P is an augmenting path with respect to M, since its end vertices are obviously exposed. □

Theorem 13.2.2 is the basis of most algorithms for determining maximal matchings in arbitrary graphs. The basic idea is obvious: we start with any given matching – for example, the empty matching or just a single edge – and try to find an augmenting path with respect to the present matching in order to enlarge the matching until no such paths exist any more. To do so, we need an efficient technique for finding augmenting paths; note that in general the number of paths in a graph grows exponentially with the size of the graph. It will be natural to use some sort of BFS starting at an exposed vertex; let us first show that no exposed vertex needs to be examined more than once.

Lemma 13.2.3. *Let G be a graph, M a matching in G, and u an exposed vertex with respect to M. Moreover, let P be an augmenting path, and put $M' = M \oplus P$. If there is no augmenting path with respect to M starting at u, then there is no augmenting path with respect to M' starting at u either.*

Proof. Let v and w be the end vertices of P; note $u \neq v, w$. Suppose there exists an augmenting path P' with respect to M' starting at u. If P and P' have no vertex in common, then P' is an augmenting path with respect to M as well, which contradicts our assumption. Thus let u' be the first vertex on P' which is contained also in P, and let e be the unique edge of M' incident with u'. Then u' divides the path P into two parts, one of which does not contain e. Let us call this part P_1, and denote the part of P' from u to u' by P_1'. Then $P_1 P_1'$ is an augmenting path with respect to M starting at u (see Figure 13.4)[2], a contradiction. □

Now suppose we have some algorithm which constructs a maximal matching step by step by using augmenting paths. We call each iteration of the algorithm in which an augmenting path is determined and used for changing the present matching according to Theorem 13.2.2 a *phase*. The following result is an immediate consequence of Lemma 13.2.3.

Corollary 13.2.4. *Assume that during some phase of the construction of a maximal matching no augmenting path starting at a given exposed vertex u exists. Then there is no such path in any of the subsequent phases either.* □

Exercise 13.2.5. Let G be a graph with $2n$ vertices, and assume either $\deg v \geq n$ for each vertex v, or $|E| \geq \frac{1}{2}(2n-1)(2n-2)+2$. Show that G has a perfect matching. Hint: Derive these assertions from a more general result involving Hamiltonian cycles.

[2] The edges of M' are drawn bold in Figure 13.4; note that $M' \cap P_1' = M \cap P_1'$ and $M' \cap P_1 \subset M \oplus P_1$.

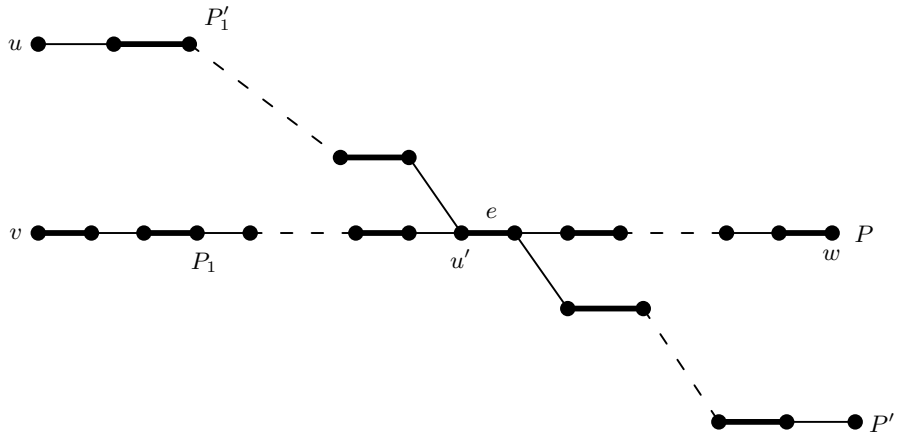


Fig. 13.4. Proof of Lemma 12.2.3

Exercise 13.2.6. Let G be a connected graph, and assume that every matching in G can be extended to a perfect matching; such a graph is called *randomly matchable*. Prove that the only randomly matchable graphs on $2n$ vertices are the graphs $K_{n,n}$ and K_{2n}; see [Sum79] and [LePP84]. Hint: Show first that G has to be 2-connected. If G is bipartite and contains non-adjacent vertices s and t which are in different parts of G, consider a path (of odd length) from s to t and construct a matching whose only exposed vertices are s and t. Finally, assume that G is not bipartite. Prove that each vertex is contained in a cycle of odd length and that any two vertices are connected by a path of odd length; then proceed as in the bipartite case.

13.3 Alternating trees and blossoms

The first polynomial algorithm for determining maximal matchings is due to Edmonds [Edm65b]; his algorithm is based on using augmenting paths according to Theorem 13.2.2. Edmonds achieved a complexity of $O(|V|^4)$ with his algorithm, although he did not state this formally. Later both Gabow [Gab76] and Lawler [Law76] proved that one may reduce the complexity to just $O(|V|^3)$ by implementing the algorithm appropriately; we shall present such a version of the algorithm in Section 13.4.

A faster (but considerably more involved) algorithm for finding maximal matchings generalizes the method of Hopcroft and Karp [HoKa73] for the bipartite case; it is due to Micali and Vazirani [MiVa80]. As in the bipartite case, this results in a complexity of $O(|V|^{1/2}|E|)$. Although an extensive discussion of this algorithm was given in [PeLo88], a formal correctness proof appeared only 14 years after the algorithm had been discovered; see Vazirani

[Vaz94]. The (theoretically) best algorithm known at present achieves a complexity of $O(|V|^{1/2}|E|\log(|V^2|/|E|)/\log|V|)$ via *graph compression*. This is due to Feder and Motwani [FeMo95] for the bipartite case; the general case is in [FrJu03]. Empirical studies concerning the quality of various algorithms for determining maximal matchings can be found in [DeHe80] and [BaDe83]; for further advances concerning implementation questions we refer to the monograph [JoMcG93].

Although it is possible to use the empty matching to initialize the construction of a maximal matching via augmenting paths, from a practical point of view it is obviously advisable to determine a reasonably large initial matching in a heuristic manner: we may expect this to result in a considerable reduction of the number of phases required by the algorithm. We will give a simple greedy method for finding such an initial matching.

Algorithm 13.3.1. Let $G = (V, E)$ be a graph with vertex set $V = \{1, \ldots, n\}$. The algorithm constructs an unextendable matching M described by an array *mate*: for $ij \in M$, $\mathrm{mate}(i) = j$ and $\mathrm{mate}(j) = i$, whereas $\mathrm{mate}(k) = 0$ for exposed vertices k. The variable *nrex* denotes the number of exposed vertices with respect to M.

Procedure INMATCH(G; mate, nrex)

(1) nrex $\leftarrow n$;
(2) **for** $i = 1$ **to** n **do** $\mathrm{mate}(i) \leftarrow 0$ **od**
(3) **for** $k = 1$ **to** $n - 1$ **do**
(4) **if** $\mathrm{mate}(k) = 0$ **and** there exists $j \in A_k$ with $\mathrm{mate}(j) = 0$
(5) **then** choose $j \in A_k$ with $\mathrm{mate}(j) = 0$;
(6) $\mathrm{mate}(j) \leftarrow k$; $\mathrm{mate}(k) \leftarrow j$; nrex $\leftarrow$ nrex -2
(7) **fi**
(8) **od**

Our next task is to design an efficient technique for finding augmenting paths; this problem turns out to be more difficult than it might appear at first sight. We begin by choosing an exposed vertex r (with respect to a given matching M of G). If there exists an exposed vertex s adjacent to r, we can extend M immediately by simply adding the edge rs. Of course, this case cannot occur if M was constructed by Algorithm 13.3.1.

Otherwise, we take r as the start vertex for a BFS and put all vertices $a_1, \ldots, a_p$ adjacent to r in the first layer; note that all these vertices are saturated. As we are looking for alternating paths, we put only the vertices $b_i = \mathrm{mate}(a_i)$ in the second layer. The next layer consists of all vertices $c_1, \ldots, c_q$ which are adjacent to one of the b_i, and where the connecting edge is not contained in M. We continue in this manner; as we will soon see, certain difficulties may arise.

If we should encounter an exposed vertex in one of the odd-numbered layers, we have found an augmenting path. This motivates the following definition: a subtree T of G with root r is called an *alternating tree* if r is an

exposed vertex and if every path starting at r is an alternating path. The vertices in layers 0, 2, 4, ... are called *outer vertices*, and the vertices in layers 1, 3, 5, ... are *inner vertices* of T.[3] Thus an alternating tree looks like the tree shown in Figure 13.5, where fat edges belong to M (as usual). Of course, the purpose of constructing an alternating tree with root r is either to find an exposed inner vertex – and thus an augmenting path – or to determine that no such vertex exists.

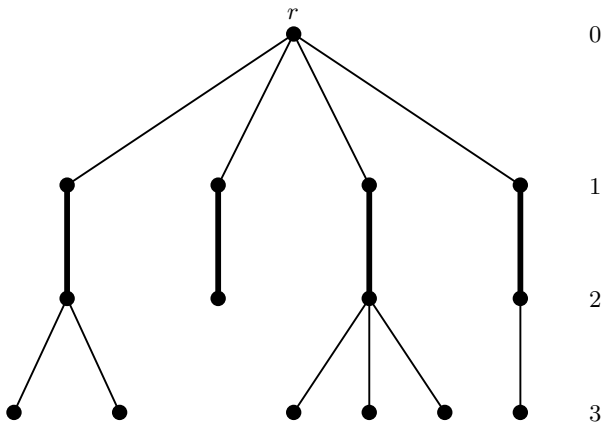


Fig. 13.5. An alternating tree

Let us suppose that the layer $2i-1$ has already been constructed. If no vertex in this layer is exposed, the next layer is easy to construct: simply add the vertex $w = \mathrm{mate}(v)$ and the edge vw to T for each vertex v in layer $2i-1$. In contrast, difficulties may arise when constructing the subsequent layer of inner vertices. Let x be a vertex in layer $2i$, and let $y \neq \mathrm{mate}(x)$ be a vertex adjacent to x. There are four possible cases.

Case 1: y is exposed (and not yet contained in T). Then we have found an augmenting path.

Case 2: y is not exposed, and neither y nor $\mathrm{mate}(y)$ are contained in T. Then we put y into layer $2i+1$ and $\mathrm{mate}(y)$ into layer $2i+2$.

Case 3: y is already contained in T as an inner vertex. Note that adding the edge xy to T would create a cycle of even length in T; see Figure 13.6. As T already contains an alternating path from r to the inner vertex y, such edges should be redundant for our purposes. We shall show later that this is indeed true, so that we may ignore this case.

[3] Some authors use the terminology *even vertex* or *odd vertex* instead.

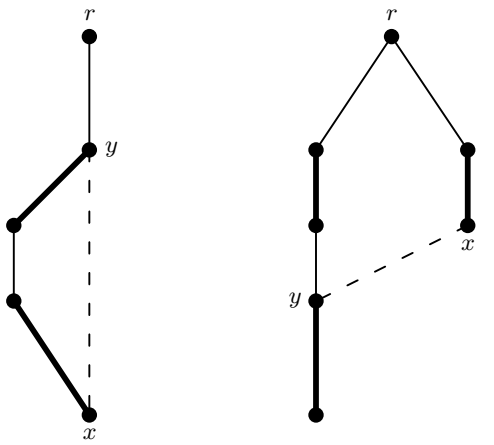


Fig. 13.6. Case 3

Case 4: y is already contained in T as an outer vertex. Note that adding the edge xy to T would create a cycle of odd length $2k+1$ in T for which k edges belong to M; see Figure 13.7.

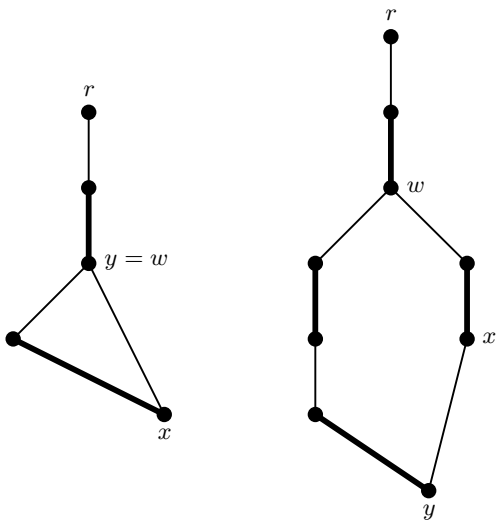


Fig. 13.7. Case 4

Such cycles are called *blossoms*; these blossoms – which of course cannot occur in the bipartite case – cause the difficulties alluded to above: edges forming a blossom with the tree constructed so far cannot just be ignored. For example,

consider the blossom displayed in Figure 13.8. Each of the vertices a, b, c, d, e, f may be reached via two different alternating paths with start vertex r: for one path, the vertex will be an outer vertex, and for the other path it will be an inner vertex. For example, a is an inner vertex with respect to (r, a), and an outer vertex with respect to (r, b, d, e, f, c, a). Now suppose that there exists an edge ax for which x is exposed; then (r, b, d, e, f, c, a, x) will be an augmenting path. If we would simply omit the edge fc when constructing T, it is quite possible that we will not find any augmenting path by our approach (even though such a path exists); the graph G in Figure 13.8 provides a simple example for this phenomenon.[4]

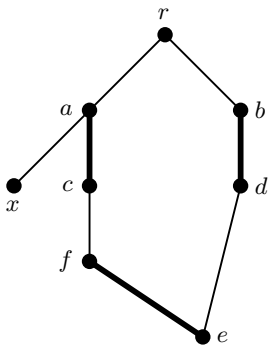


Fig. 13.8. A blossom

The difficulties arising from Case 4 are avoided in the algorithm of Edmonds by *shrinking blossoms* to single vertices. At a later point of the algorithm, blossoms which were shrunk earlier may be *expanded* again. We shall treat this process in the next section.

In the bipartite case, constructing an alternating tree T presents no problems, as no cycles of odd length can occur. Hence there are no blossoms, and it is clear for all vertices whether they have to to be added as inner or as outer

[4] It is tempting to proceed by using all vertices of a blossom both as inner and as outer vertices, so that we cannot miss an augmenting path which uses part of a blossom. Indeed, Pape and Conradt [PaCo80] proposed splitting up each blossom into two alternating paths, so that the vertices of a blossom appear twice in the alternating tree T, both as inner and as outer vertices. Unfortunately, a serious problem arises: it might happen that an edge xy which was left out earlier in accordance with Case 3 (that is, an edge closing a cycle of even length) is needed at a later point because it is also contained in a blossom. The graph shown in Figure 13.9 contains a unique augmenting path (between r and r') which is not detected by the algorithm of [PaCo80]; thus their algorithm is too simple to be correct! This counterexample is due to Christian Fremuth-Paeger; see the first edition of the present book for more details.

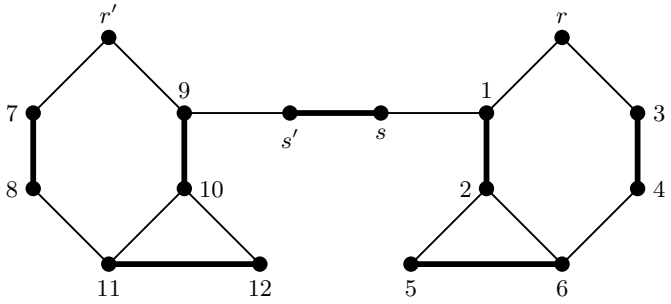


Fig. 13.9. A graph with a unique augmenting path

vertices to T: with $V = S \dot\cup S'$ and $r \in S$ (say), all vertices of S which are accessible from r have to be outer vertices, and all vertices of S' accessible from r have to be inner vertices. Thus there exists an augmenting path starting at r if and only if the corresponding alternating tree contains an exposed inner vertex. For the sake of completeness, we now present an algorithm for constructing a maximal matching in a bipartite graph which uses this technique. Even though its complexity is worse than the complexity guaranteed by Theorem 7.2.1, it is of some interest to have a method (which is, after all, still quite good) which does not depend on network flows. Moreover, the algorithm in question will also be the basis of the Hungarian algorithm to be treated in Chapter 14.

Algorithm 13.3.2. Let $G = (V, E)$ be a bipartite graph with respect to the partition $V = S \dot\cup S'$, where $S = \{1, \ldots, n\}$ and $S' = \{1', \ldots, m'\}$; we assume $n \leq m$. The algorithm constructs a maximal matching M described by an array *mate*. The function $p(y)$ gives, for $y \in S'$, the vertex in S from which y was accessed.

Procedure BIPMATCH(G; mate, nrex)

```
(1) INMATCH(G; mate, nrex);
(2) r ← 0;
(3) while nrex ≥ 2 and r < n do
(4)     r ← r + 1;
(5)     if mate(r) = 0
(6)     then for i = 1 to m do p(i′) ← 0 od
(7)         Q ← ∅; append r to Q; aug ← false;
(8)         while aug = false and Q ≠ ∅ do
(9)             remove the first vertex x of Q;
(10)            if there exists y ∈ A_x with mate(y) = 0
(11)            then choose such a y;
(12)                while x ≠ r do
(13)                    mate(y) ← x; next ← mate(x); mate(x) ← y;
```

```
(14)                    y ← next; x ← p(y)
(15)                od
(16)                mate(y) ← x; mate(x) ← y; nrex ← nrex−2; aug ← true
(17)            else for y ∈ A_x do
(18)                if p(y) = 0 then p(y) ← x; append mate(y) to Q fi
(19)                od
(20)            fi
(21)        od
(22)    fi
(23) od
```

Of course, it is also possible to use the empty matching for initializing the construction: simply replace (1) and (2) by

(1′) **for** $v \in V$ **do** mate$(x) \leftarrow 0$ **od**
(2′) $r \leftarrow 0$; nrex $\leftarrow n$;

We leave it to the reader to prove the following result.

Theorem 13.3.3. *Let $G = (V, E)$ be a bipartite graph with respect to the partition $V = S \dot\cup S'$. Then Algorithm 13.3.2 determines with complexity $O(|V||E|)$ a maximal matching of G.* □

Balinsky and Gonzales [BaGo91] gave an algorithm for determining a maximal matching of a bipartite graph which does not rely on augmenting paths; their algorithm also has complexity $O(|V||E|)$.

13.4 The algorithm of Edmonds

In this section, $G = (V, E)$ is always a connected graph with a given initial matching M; we present the algorithm for constructing maximal matchings due to Edmonds [Edm65b]. We begin by constructing an alternating tree T with root r, as described in the previous section. Edges xy closing a cycle of even length (Case 3 in Section 13.3) will be ignored. Whenever we encounter an edge xy closing a blossom B (Case 4 in Section 13.3, see Figure 13.7), we stop the construction of T and *shrink* the blossom B. Formally, we may describe this operation as *contracting* G with respect to B to a smaller graph G/B which is defined as follows:

- The vertex set of G/B is $V/B = (V \setminus B) \cup \{b\}$, where b is a new vertex (that is, $b \notin V$ is a new symbol).
- The edge set E/B of G/B is derived from E by first removing all edges $uv \in E$ with $u \in B$ or $v \in B$ and then adding an edge ub for all those $u \in V \setminus B$ which are adjacent in G to at least one vertex of B.[5]

[5] Note that G/B is the result of a sequence of elementary contractions with respect to the edges contained in the blossom B; see Section 1.5.

To distinguish it from the original vertices, the new vertex b is called a *pseudovertex* of G/B. Now we have to address the question how shrinking a blossom B effects the construction of T. Note that the matching M of G induces in a natural way a matching M/B of G/B. When we encounter an edge xy closing a blossom B, we know just two vertices of B, namely x and y. The whole blossom can be determined by following the paths from x and y to the root r in T; the first common vertex w of these two paths is called the *base* of the blossom B. Note that w is an outer point of T. Then B is the union of xy with the two paths P_{wx} and P_{wy} from the base w to the vertices x and y, respectively. Omitting these two paths from T and replacing the base w by the pseudovertex b yields an alternating tree T/B for G/B with respect to the matching M/B. Now we proceed with our construction in G/B, using T/B; here the next outer vertex we examine is the pseudovertex b. Of course, further blossoms may arise, in which case we will have to perform a series of shrinkings. Let us illustrate this procedure with an example.

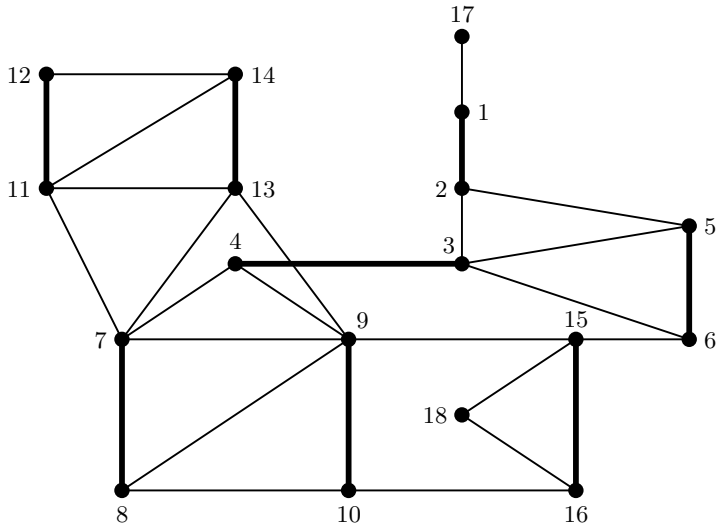


Fig. 13.10. A graph G with an initial matching M

Example 13.4.1. Let G be the graph shown in Figure 13.10. Starting at the vertex $r = 17$, we construct the alternating tree T displayed in Figure 13.11. We examine outer vertices in the order in which they were reached (that is, we use a BFS-type ordering) and use increasing order for the adjacency lists. Note that the edge $\{6, 3\}$ is ignored (Case 3 in Section 13.3) when the outer vertex 6 is examined, because it closes a cycle of even length. Similarly, the edge $\{8, 9\}$ is ignored when we examine the vertex 8.

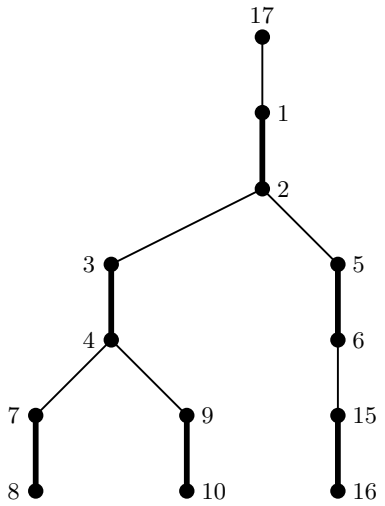


Fig. 13.11. Alternating tree T in G

The next edge $\{8, 10\}$ closes a blossom, namely $B = \{4, 7, 8, 9, 10\}$. This blossom has base 4 and is contracted to a pseudovertex b; we obtain the graph $G' = G/B$ with the matching $M' = M/B$ and the corresponding alternating tree T/B shown in Figure 13.12.

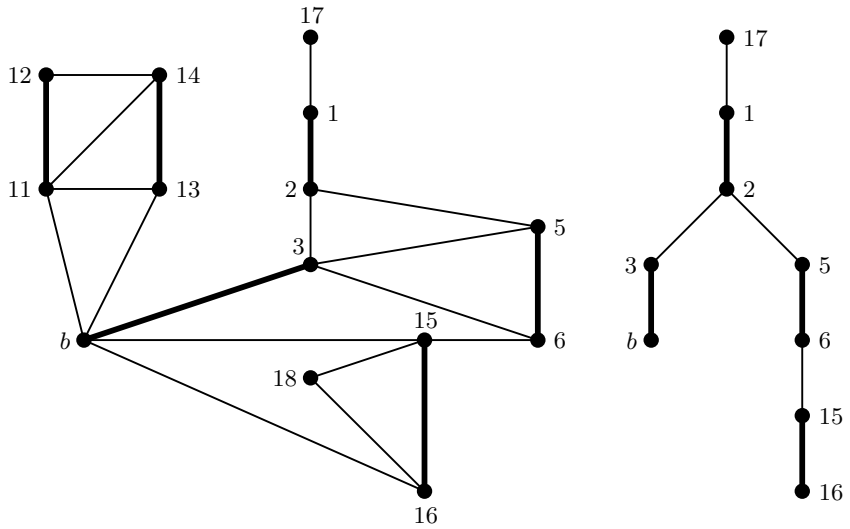


Fig. 13.12. Contracted graph G/B with alternating tree T/B

Continuing the construction with the outer vertex b (which is nearest to the root $r = 17$), we obtain the alternating tree T' in Figure 13.13. Here the edge $\{b, 15\}$ is ignored in accordance with Case 3, whereas the edge $\{b, 16\}$ closes a further blossom $B' = \{b, 2, 3, 5, 6, 15, 16\}$ with base 2; thus B' has to be contracted to a second pseudovertex b'. Note that pseudovertices may contain other pseudovertices which were constructed earlier. The result is the graph $G'' = G'/B'$ with the matching $M'' = M'/B'$ and the corresponding tree $T'' = T'/B'$ in Figure 13.14.

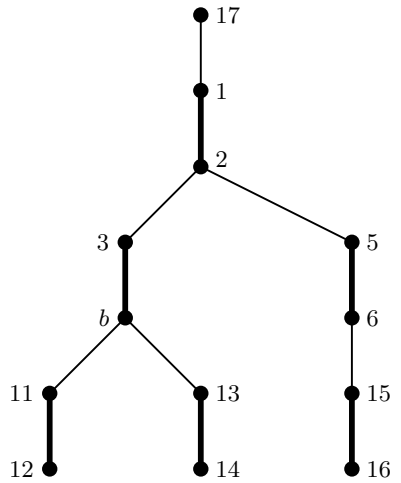


Fig. 13.13. Alternating tree T' for G'

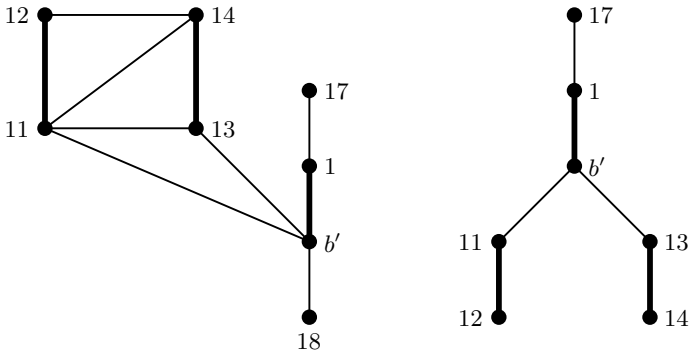


Fig. 13.14. Contracted graph G'/B' with alternating tree T'/B'

Now the outer vertex b' is examined. This time we find an adjacent exposed vertex, namely 18, which yields the augmenting path P'': $18 — b' — 1 — 17$ in G''. We want to use this path to determine an augmenting path P in G. For this purpose, we trace P'' backwards from its end vertex 18 to the root 17 of T''. The first vertex we reach is the pseudovertex b'; thus there has to be at least one vertex p in G' which is adjacent to 18 and contained in the blossom B'. In fact, there are two such vertices: 15 and 16. We choose one of them, say $p = 15$. In the blossom B', p is incident with a unique edge of M', namely the edge $e = \{15, 16\}$. We trace the path from 15 to 16 and continue in B' until we reach the base 2 of B'.

Thus we have constructed an augmenting path P' in G' from the augmenting path P'' in G'':

$$P' : \ 18 — 15 — 16 — b — 3 — 2 — 1 — 17$$

Similarly, we encounter the pseudovertex b when we trace P' backwards from its end vertex 18 to the root 17 of T'. Thus 16, the immediate predecessor of b on P', has to be adjacent to at least one vertex of the blossom B; this vertex is 10. The unique edge of M incident with 10 is $\{10, 9\}$, so that we traverse the blossom B from 10 to 9 and on to its base 4. This yields the desired augmenting path in our original graph G:

$$P : \ 18 — 15 — 16 — 10 — 9 — 4 — 3 — 2 — 1 — 17$$

Finally, we augment our initial matching M using the path P, which yields the perfect matching shown in Figure 13.15.

Exercise 13.4.2. Use the method described in Example 13.4.1 to enlarge the matching shown in the graph of Figure 13.9. Take r as the root of the alternating tree; if choices have to be made, use the vertices according to increasing labels.

Hint: You can simplify this task by exploiting the inherent symmetry of the graph in question, which allows you to consider its two halfs separately.

The algorithm of Edmonds generalizes the method used in Example 13.4.1. Before stating his algorithm explicitly, we ought to prove that the shrinking process for blossoms always works correctly. It will suffice to show that the graph $G' = G/B$ resulting from contracting the first blossom B which we encounter contains an augmenting path with start vertex r (or b, if r should be contained in B) if and only if the original graph G contains such a path. (In the case $r \in B$, the vertex r of G is replaced with the pseudovertex b in G' by the shrinking process.) Then the general result follows by induction on the number of blossoms encountered. We will prove our assertion by establishing the following two lemmas.

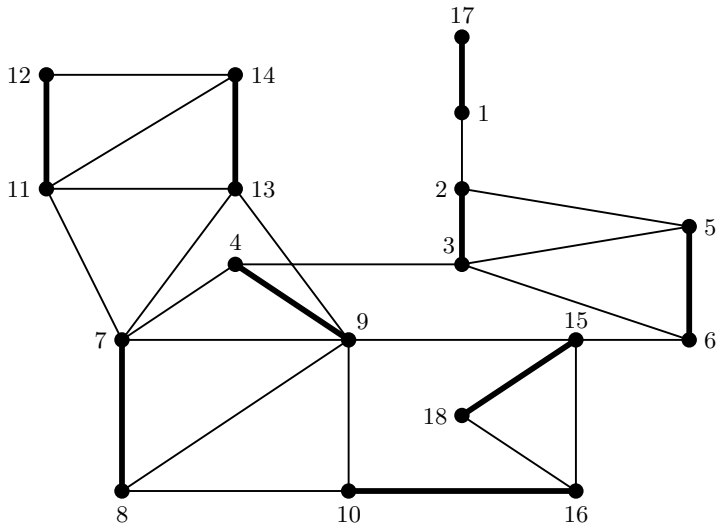


Fig. 13.15. A perfect matching in G

Lemma 13.4.3. *Let G be a connected graph with a matching M, and let r be an exposed vertex with respect to M. Suppose that, during the construction of an alternating tree T with root r (according to the rules described above), the first blossom B is found when the edge $e = xy$ is examined; here x denotes the outer vertex which the algorithm examines at this time, and y is another outer vertex of T. Let w be the base of B, and consider the contracted graph $G' = G/B$ which results by replacing B with the pseudovertex b. If G contains an augmenting path with respect to M starting at r, then G' contains an augmenting path with respect to the induced matching $M' = M/B$ starting at r (or at b, when $r \in B$).*

Proof. Let P be an augmenting path in G with respect to M starting at r, and denote the end vertex of P by s. As all vertices in P are saturated except for r and s, we may actually assume that r and s are the only exposed vertices of G. (Otherwise, we may remove all further exposed vertices together with the edges incident with them from G.) Assume first that P and B do not have any vertices in common. Then the assertion is obvious: P is also an augmenting path in G' with respect to M'. Thus we may assume that P and B are not disjoint. We distinguish two cases.

Case 1: The root r is contained in the blossom B, so that r is the base of B. We trace P from r to s. Let q be the first vertex of P which is not contained in B, and denote its predecessor on P by p; thus p is the last vertex of P contained in B. (Note that $p = r$ is possible.) Then the edge pq is not contained in B. Denote the part of P from r to p by P_1, and the part from q to s by P_2; see

Figure 13.16. Clearly, $P' = b \text{—} q \overset{P_2}{\text{—}} s$ is an augmenting path in G' with respect to M'.

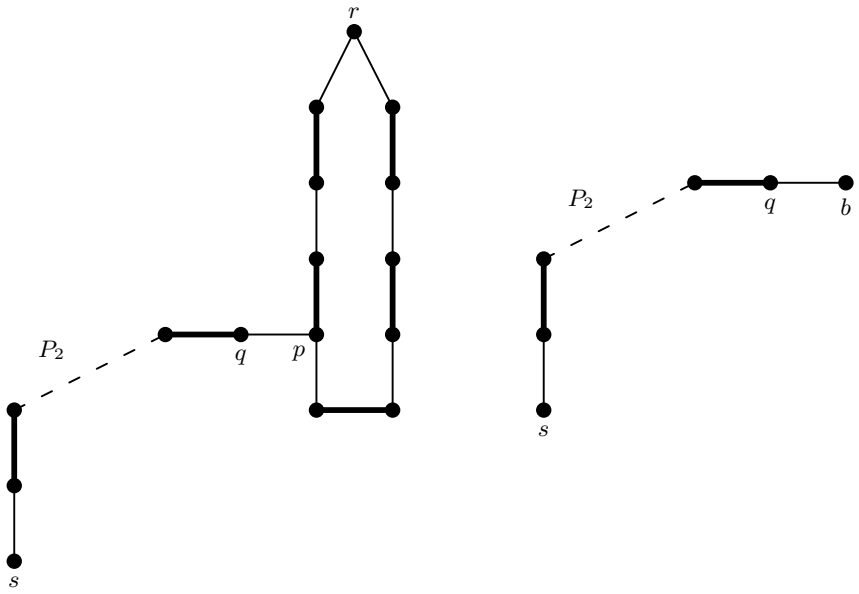


Fig. 13.16. Case 1 of Lemma 13.4.3

Case 2: The root r of T is not contained in B, so that the base of B is an outer vertex $w \neq r$. Denote the alternating path of even length from r to w in T by S; S is usually called the *stem* of the blossom B. This time it is not quite obvious how the augmenting path P with respect to M interacts with the blossom B. Therefore we will use a trick which allows us to reduce this case to Case 1: we replace M by the matching $M_1 = M \oplus S$, which has the same cardinality. Then w and s are the only exposed vertices with respect to M_1, so that the blossom B (which has not been changed) has base w if we begin constructing an alternating tree at vertex w; see Figure 13.17. Thus the situation for M_1 is really as in Case 1.

As there exists an augmenting path in G with respect to M, M was not maximal. Hence M_1 is not maximal either; by Theorem 13.2.2, there exists an augmenting path P_1 with respect to M_1 in G. According to Case 1, this implies the existence of an augmenting path P_1' in G' with respect to M_1/B, so that the matching M_1/B is not maximal. It follows that the matching M/B of G' (which has the same cardinality as the matching M_1/B) is not maximal either; hence there must be an augmenting path in G' with respect to M/B. As r and s are the only exposed vertices in G', the assertion follows. □

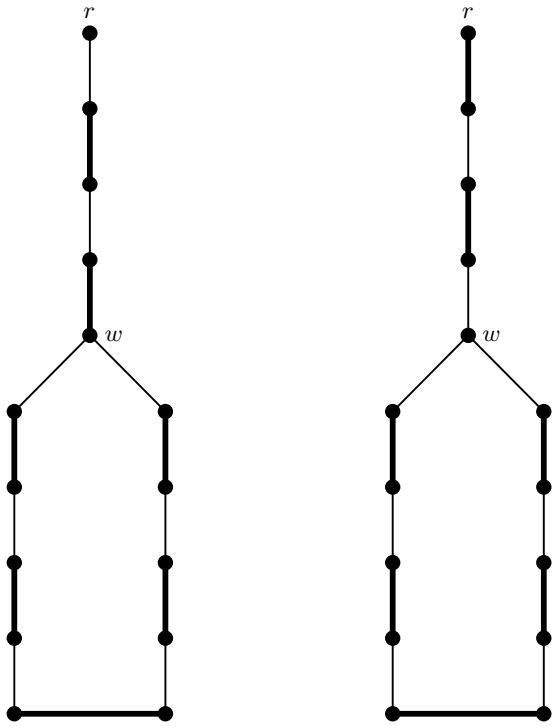


Fig. 13.17. Case 2 of Lemma 13.4.3

Lemma 13.4.4. *Let G be a connected graph with a matching M, and suppose that r is an exposed vertex with respect to M. Moreover, let B be a blossom with base w and $G' = G/B$ the contracted graph where B is replaced with the pseudovertex b. If G' contains an augmenting path with respect to $M' = M/B$ starting at r (or at b, when $r \in B$), then there exists an augmenting path in G with respect to M starting at r.*

Proof. Assume first that the augmenting path P' in G' does not contain the pseudovertex b. Then P' is also a path in G, and the assertion is clear. Thus suppose $b \in P'$. We consider only the case $r \notin B$; the case $r \in B$ is similar and actually even simpler. Let w be the base of B. First suppose that the distance from r to b in P' is even. Then P' has the form

$$P' : \; r \xrightarrow{P_1} p - b - q \xrightarrow{P_3} s,$$

where P_1 is the part of P' from r to $p = \mathrm{mate}(b)$. Now q must be adjacent to a vertex $q' \in B$. Denote the alternating path of even length in B from w to q' by P_2; note $P_2 = \emptyset$ if $w = q'$. Then

$$P : \; r \xrightarrow{P_1} p - w \xrightarrow{P_2} q' - q \xrightarrow{P_3} s$$

is an augmenting path in G with respect to M, where P_3 denotes the part of P' from q to s; see Figure 13.18.

Finally, if the distance from r to b in P' is odd, the distance from s to b in P' has to be even. Then we simply exchange the roles of r and s and proceed as before. (This case may indeed occur, as the solution to Exercise 13.4.2 shows.) □

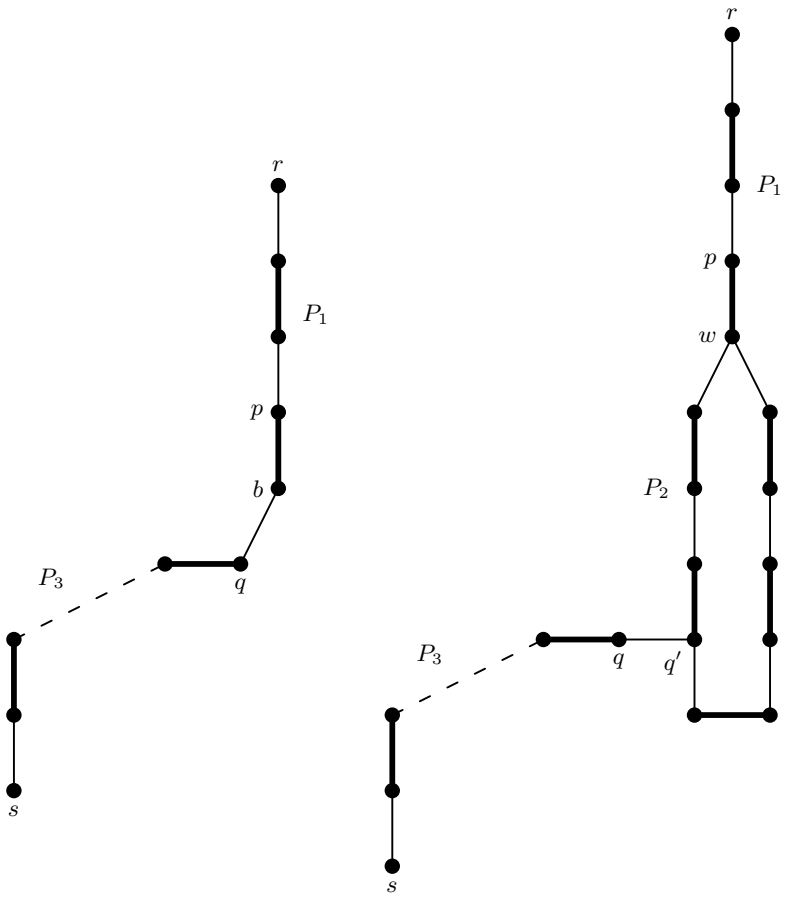


Fig. 13.18. Proof of Lemma 13.4.4

As the following exercise shows, the condition that the blossom B under consideration must have been found during the construction of the alternating tree T is actually needed in Lemma 13.4.3, whereas no such condition on B is required for Lemma 13.4.4.

Exercise 13.4.5. Consider the graph G with the matching M shown in Figure 13.19. Obviously, G contains a unique blossom. Show that the contracted

graph G' does not contain an augmenting path with respect to M', even though G contains an augmenting path with respect to M.

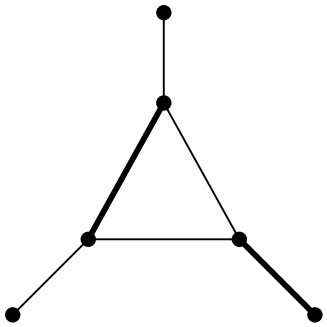


Fig. 13.19. Graph with a blossom

Now we are ready to state a version of the algorithm of Edmonds, which more or less generalizes the method used in Example 13.4.1. There will be one major difference, though: in order to achieve a better complexity, the graph will not be contracted explicitly when a blossom B is encountered, as this would require rather involved update operations and also a later re-expansion of the contracted graph. Instead, the vertices in B will be declared *inactive*, which is done with the help of a Boolean function $a(v)$ on V.

Algorithm 13.4.6 (Algorithm of Edmonds). Let $G = (V, E)$ be a graph on the vertex set $V = \{1, \ldots, n\}$, given by adjacency lists A_v. The algorithm constructs a maximal matching M of G described by an array *mate* and determines the number of exposed vertices of G with respect to M.

The main procedure MAXMATCH uses the procedure INMATCH given in Algorithm 13.3.1 to determine an initial matching as well as three further auxiliary procedures: BLOSSOM, CONTRACT, and AUGMENT. These three procedures are described after MAXMATCH in a less formal way. In MAXMATCH, the function d describes the position of the vertices in the current alternating tree T with root r: vertices which are not yet in the tree have $d(y) = -1$; for all other vertices, $d(y)$ is the distance between y and the root r of T. In particular, vertices y for which $d(y)$ is odd are inner vertices, and vertices y for which $d(y)$ is even are outer vertices. The outer vertices are kept in a priority queue Q with priority function d.

The construction of the alternating tree T is always continued from the first active vertex of Q. Initially, all vertices are active. Vertices become inactive if they are contained in a blossom which is contracted. The examination of the neighbors of an outer vertex x is done as described in Section 13.3, and blossoms are contracted immediately when they are discovered.

As we need the original adjacency lists A_v of G later for expanding the augmenting paths (as in Example 13.4.1), these lists must not be changed throughout the algorithm. Therefore we will use new adjacency lists $CA(v)$ for describing the contracted graphs. As mentioned before, the vertices of a contracted blossom are not actually removed from the graph, but are just declared to be *inactive*; of course, originally all vertices are active. For this purpose, we use a Boolean function a: a vertex v remains active as long as $a(v)$ has the value *true*.

Finally, there are also Boolean variables *aug* and *cont*, which serve to control the loop: the variable *aug* has the value *false* until an augmenting path is found; and during the examination of an outer vertex x, *cont* has value *false* until a blossom is found (and contracted).

Procedure MAXMATCH(G; mate, nrex)

```
(1) INMATCH(G; mate, nrex);
(2) r ← 0;
(3) while nrex ≥ 2 and r < n do
(4)     r ← r + 1;
(5)     if mate(r) = 0
(6)     then Q ← ∅, aug ← false, m ← 0;
(7)         for v ∈ V do
(8)             p(v) ← 0, d(v) ← −1, a(v) ← true;
(9)             CA(v) ← A_v
(10)        od
(11)        d(r) ← 0; append r to Q;
(12)        while aug=false and Q ≠ ∅ do
(13)            remove the first vertex x of Q;
(14)            if a(x) = true
(15)            then cont ← false
(16)                for y ∈ CA(x) do u(y) ← false od
(17)                repeat
(18)                    choose y ∈ CA(x) with u(y) = false; u(y) ← true;
(19)                    if a(y) = true
(20)                    then if d(y) ≡ 0 (mod 2)
(21)                        then m ← m + 1;
(22)                            BLOSSOM (x, y; B(m), w(m));
(23)                            CONTRACT (B(m), m, w);
(24)                        else if d(y) = −1
(25)                            then if mate(y) = 0
(26)                                then AUGMENT(x, y)
(27)                                else z ← mate(y);
(28)                                    p(y) ← x; d(y) ← d(x) + 1;
(29)                                    p(z) ← y; d(z) ← d(y) + 1;
(30)                                    insert z with priority d(z) into Q
(31)                                fi
```

```
(32)                              fi
(33)                           fi
(34)                        fi
(35)                     until u(y) = true for all y ∈ CA(v) or aug = true
                         or cont = true
(36)                  fi
(37)               od
(38)         fi
(39) od
```

The following procedure BLOSSOM constructs a blossom B with base w. This procedure is called by MAXMATCH if a further outer vertex y is discovered in $CA(x)$ during the examination of an outer vertex x (according to Case 4 in Section 13.3).

Procedure BLOSSOM$(x, y; B, w)$

```
(1) P ← {x}; P' ← {y}; u ← x; v ← y;
(2) repeat
(3)       P ← P ∪ {p(u)}; u ← p(u)
(4) until p(u) = r;
(5) repeat
(6)       P' ← P' ∪ {p(v)}; v ← p(v)
(7) until v = r;
(8) S ← P ∩ P';
(9) let w be the element of S for which d(w) ≥ d(z) for all z ∈ S;
(10) B ← ((P ∪ P') \ S) ∪ {w}
```

The procedure CONTRACT is used for contracting a blossom B, and the adjacency lists $CA(v)$ for the contracted graph are updated accordingly.

Procedure CONTRACT(B, m, w)

```
(1) b ← n + m; a(b) ← true;
(2) p(b) ← p(w); d(b) ← d(w); mate(b) ← mate(w);
(3) insert b into Q with priority d(b);
(4) CA(b) ← ⋃_{z∈B} CA(z);
(5) for z ∈ CA(b) do CA(z) ← CA(z) ∪ {b} od
(6) for z ∈ B do a(z) ← false od
(7) for z ∈ CA(b) do
(8)       if a(z) = true and p(z) ∈ B
(9)       then d(z) ← d(b) + 1; p(z) ← b;
(10)            d(mate(z)) ← d(z) + 1;
(11)      fi
(12) od
(13) cont ← true
```

The final procedure AUGMENT serves to construct an augmenting path (and to change the matching M accordingly) when an exposed vertex y is encountered during the construction of the alternating tree T; see step (25) in MAXMATCH.

Procedure AUGMENT(x, y)

(1) $P \leftarrow \{y, x\}$; $v \leftarrow x$;
(2) **while** $p(v) \neq 0$ **do**
(3) $\quad P \leftarrow P \cup \{p(v)\}$; $v \leftarrow p(v)$
(4) **od**
(5) **while** there exists $b \in P$ with $b > n$ **do**
(6) $\quad$ choose the largest $b \in P$ with $b > n$;
(7) $\quad B \leftarrow B(b-n)$; $w \leftarrow w(b-n)$; $z \leftarrow \text{mate}(w)$;
(8) $\quad$ let q be the neighbor of b on P which is different from z;
(9) $\quad$ choose some $q' \in B \cap CA(q)$;
(10) $\quad$ determine the alternating path B' of even length in B from w to q';
(11) $\quad$ replace b by w in P;
(12) $\quad$ insert B' into P between w and q
(13) **od**
(14) $u \leftarrow y$; $v \leftarrow x$;
(15) **while** $v \neq r$ **do**
(16) $\quad z \leftarrow \text{mate}(v)$; $\text{mate}(v) \leftarrow u$; $\text{mate}(u) \leftarrow v$;
(17) $\quad u \leftarrow z$; let v be the successor of z on P
(18) **od**
(19) $\text{mate}(v) \leftarrow u$; $\text{mate}(u) \leftarrow v$;
(20) nrex $\leftarrow$ nrex -2; aug $\leftarrow$ true

Theorem 13.4.7. *Let $G = (V, E)$ be a connected graph. Then Algorithm 13.4.6 determines a maximal matching of G. If the auxiliary procedures BLOSSOM, CONTRACT, and AUGMENT are implemented appropriately, one may achieve an overall complexity of $O(|V|^3)$.*

Proof. The detailed discussion given when we derived Edmonds' algorithm in the present and the previous sections already shows that Algorithm 13.4.6 is correct. Nevertheless, we will summarize the main points once again.

First, an initial matching M described by the array *mate* is constructed via the procedure INMATCH. The subsequent outer **while**-loop in MAXMATCH comprises the search for an augmenting path with respect to M with start vertex r: it constructs an alternating tree T with root r. Obviously, this search can only be successful if there are still at least two exposed vertices; hence we require nrex ≥ 2. Moreover, we may restrict to $r \leq n-1$ the examination of exposed vertices r as start vertices for an augmenting path, because an augmenting path with start vertex n would have been found earlier when its end vertex was used as the root of an alternating tree. It follows from

Theorem 13.2.2 and Lemma 13.2.3 that it indeed suffices to examine each exposed vertex at most once as the root of an alternating tree.

As already mentioned, the outer vertices of T are kept in the priority queue Q and examined in a BFS-like fashion (provided that they are still active). During the inner **while**-loop, the first active vertex x from Q is chosen – as long as this is possible and no augmenting path has been found yet. By examining the vertices adjacent to x, the construction of the tree T is continued. Choosing x according to the priority function $d(x)$ ensures that the construction of T is always continued from a vertex which has smallest distance to the root r.[6] Inner vertices y are ignored during the examination of the vertices adjacent to x according to the conditions in steps (20) and (24): in that case, the edge xy would close a cycle of even length, and y would already be accessible from r by an alternating path of odd length in T. Note that the function $d(y)$ is used to decide whether y is already contained in T: either $d(y) > 0$, and this value equals the distance between r and y in T; or y has not been added to the tree yet, in which case $d(y) = -1$.

If the condition on $d(y)$ holds in step (20) (so that $d(y)$ is even and hence y is contained in T as an outer vertex), the edge xy closes a blossom B. According to Lemmas 13.4.3 and 13.4.4, we may then continue with the contracted graph G/B instead of G: there is an augmenting path with respect to M in G if and only if G/B contains an augmenting path with respect to the induced matching M/B. The first step in replacing G by G/B is to construct the blossom B by calling the procedure BLOSSOM in step (22). This procedure uses the *predecessor function* p defined in steps (28) and (29) of the main procedure: $p(v)$ is the predecessor of v on the unique path from the root r to v in T. With the help of p, BLOSSOM determines the paths P and P' from x and y, respectively, to the root r in the obvious manner. Clearly, the intersection S of P and P' is the stem of B. As the function d gives the distance of a vertex from the root r of T, the base of B is precisely the vertex w of S for which $d(w)$ is maximal. Therefore the blossom B is indeed $((P \cup P') \setminus S) \cup \{w\}$ as stated in step (10) of BLOSSOM; thus this procedure indeed constructs the blossom B as well as its base w.

Next, the procedure CONTRACT is called in step (23) of MAXMATCH in order to replace the graph G with the contracted graph G/B and to change M and T accordingly. The pseudovertex b to which B is contracted is numbered as $b = n + m$, where m counts the number of blossoms discovered up to this point (including B). This makes it easy to decide which vertices of a contracted graph are pseudovertices: the pseudovertices are precisely those vertices b with $b > n$. Steps (1) to (3) of CONTRACT label b as an active vertex, insert it into Q, and replace the base w and all other vertices of B

[6] In contrast to an ordinary BFS, we need the explicit distance function $d(x)$ because contractions of blossoms may change the distances in the current tree T: a new pseudovertex b is, in general, closer to the root than some other active vertices which were earlier added to T; compare Example 13.4.1.

by b in T: the predecessor of b is defined to be $p(w)$, and the distance of b from r is set to $d(w)$; moreover, mate(b) is defined to be mate(w), as we also need the induced matching M/B. Steps (4) to (6) contain the implicit contraction: all vertices of B are labelled as inactive and all vertices adjacent to some vertex of B are made neighbors of b by putting them into $CA(b)$. In steps (7) to (12), T is updated to T/B by defining b to be the predecessor of all active vertices z whose predecessor $p(z)$ was some vertex in B, and by defining the distance of these vertices to the root r to be $d(b)+1$. The same is done for the corresponding outer vertices mate(z). Finally, the variable *cont* is assigned the value *true*; this means that the examination of the vertex x (which is no longer active) is stopped in MAXMATCH, and the construction of T is continued with the active vertex in Q which has highest priority (that is, smallest distance from r).

If a vertex y is not yet contained in T when the edge xy is examined (that is, if $d(y) = -1$), we check in step (25) whether y is exposed. If this is not the case, the vertices y and $z = \text{mate}(y)$ are added to the tree T (as an inner and an outer vertex, respectively) by appending the path $x - y - z$ at x, and by defining the predecessors and the distances of y and z to r accordingly in steps (28) and (29); also, the new active outer vertex z is inserted into Q with priority $d(z)$ in step (30). Finally, if y is exposed, AUGMENT replaces M with a larger matching according to Theorem 13.2.2.

Steps (1) to (4) of the procedure AUGMENT construct an augmenting path P with respect to the present matching M' which is induced by M in the graph G' (the current, possibly multiply contracted graph). During the first **while**-loop, the pseudovertices on this path (which are recognized by the condition $b > n$) are *expanded* in decreasing order: the pseudovertices which were constructed first (and thus have smaller labels) are expanded last. To execute such an *expansion*, the neighbor $q \neq \text{mate}(b)$ of b is determined and the edge $\{b, q\}$ on P is replaced by the alternating path of even length from the base w of the corresponding blossom to the vertex $q' \in B$ adjacent to q; compare the proof of Lemma 13.4.4. Therefore P is an augmenting path in G with respect to M when the first **while**-loop is terminated; we view P as being oriented from y to the root r. The second **while**-loop augments the matching M along this path by updating the function *mate* appropriately. In step (20) of AUGMENT, the number of exposed vertices is decreased by 2 and the variable aug is assigned value *true*; hence the construction of the tree T is stopped in MAXMATCH according to step (12) or (35). Then the outer **while**-loop of MAXMATCH starts once again, using the next exposed vertex as the root r of a new alternating tree (if possible).

The **repeat**-loop in MAXMATCH (which comprises the search for an exposed outer vertex from a fixed vertex x) is terminated according to step (35) if either a contraction or an augmentation has been performed, or if all currently active vertices y adjacent to x have been examined. The inner **while**-loop terminates if either an augmenting path was found or if Q is empty; in the latter case the construction of T terminates without finding an

augmenting path. In this case, we have constructed an alternating tree T' for a (in general, multiply contracted) graph G' in which all blossoms discovered during the construction of the tree with root r were immediately contracted. Therefore, as there is no augmenting path with start vertex r in G', there is no such path in G either – by Lemma 13.4.4. In both cases, the algorithm continues with the next exposed vertex as the root of an alternating tree. This process continues until the outer **while**-loop is terminated (as discussed above), so that the then current matching M is maximal.

It remains to discuss the complexity of MAXMATCH. Obviously, there are at most $O(|V|)$ iterations of the outer **while**-loop; in other words, the algorithm has at most $O(|V|)$ phases (where an alternating tree with root r is constructed). Hence we want to show that each phase can be executed in $O(|V|^2)$ steps, provided that the auxiliary procedures are implemented suitably.

Note that each blossom contains at least three vertices and that each vertex may be contracted (that is, made inactive) at most once, so that there are only $O(|V|)$ contractions during a given phase. To be more precise, at most $|V|/2$ pseudovertices may occur during the inner **while**-loop, which implies that there are at most $O(|V|)$ iterations of this loop. When the vertices y adjacent to x are examined during the **repeat**-loop, at most $O(|V|)$ edges are treated, so that this whole process can take at most $O(|V|^2)$ steps – not counting the complexity of the auxiliary procedures BLOSSOM, CONTRACT, and AUGMENT. It is easily seen that one call of BLOSSOM takes at most $O(|V|)$ steps; since there are at most $O(|V|)$ such calls in a phase, these operations also contribute at most $O(|V|^2)$ steps.

We next show that the calls of CONTRACT during a given phase altogether need at most $O(|V|^2)$ steps, provided that step (4) is implemented appropriately: note that the construction of the adjacency list of the pseudovertex b is the only part of CONTRACT whose complexity is not quite obvious. Fortunately, it is possible to perform this construction efficiently by using a labelling process: initially, all vertices are unlabeled; then we label all vertices occurring in one of the adjacency lists of the vertices contained in B; and finally, we define $CA(b)$ to be the set of all labelled vertices. For each blossom B, this labelling method requires $O(|V|)$ steps plus the number of steps we need for examining the adjacency lists of the vertices of B. Now there are only $O(|V|)$ vertices which might occur in one of the blossoms and need to be examined then; hence these examinations – added over all calls of CONTRACT – cannot take more than $O(|V|^2)$ steps altogether.

Finally, we have to convince ourselves that an eventual call of AUGMENT has complexity at most $O(|V|^2)$. Obviously, there are at most $O(|V|)$ iterations of the first **while**-loop in AUGMENT. All operations during this loop can be executed in $O(|V|)$ steps, except possibly for the determination of a vertex q' in $CA(q)$ in step (9) and the determination of an alternating path from w to q' in step (10) during the expansion of a pseudovertex b to a blossom B. However, the first of these two operations may be implemented via a labelling

process, which easily leads to a complexity of $O(|V|)$: we label the vertices in $CA(q)$ and then look for a labelled vertex in B. The second operation may also be performed in $O(|V|)$ steps if we store the blossom B in BLOSSOM not just as a set, but as a doubly linked list: then we simply trace B from q' in the direction given by mate(q') until we reach the base w. Therefore, AUGMENT likewise allows a complexity of $O(|V|^2)$.

Summing up, each phase of MAXMATCH may be performed in $O(|V|^2)$ steps, so that the overall complexity is indeed $O(|V|^3)$. □

It should be mentioned that actually implementing the procedures for determining a maximal matching (as described above) is in fact a rather daunting task. We close this section with an exercise.

Exercise 13.4.8. Let G be a bipartite graph with vertex set $V = S \dot\cup T$, where $S = \{1, \ldots, n\}$ and $T = \{1', \ldots, n'\}$. G is called *symmetric* if the existence of an edge ij' in G always implies that also ji' is an edge of G. A matching of G is called *symmetric* if M does not contain any edge of the form ii' and if, for each edge $ij' \in M$, the edge ji' is contained in M as well. How could a maximal symmetric matching in a symmetric bipartite graph be determined?

13.5 Matching matroids

In this final section we return to theoretical considerations once again. We will present the generalization of Theorem 7.3.8 due to Edmonds and Fulkerson [EdFu65] which was already mentioned in Section 7.3.

Theorem 13.5.1. *Let $G = (V, E)$ be a graph, and let $\mathbf{S}$ be the set of all those subsets of vertices which are covered by some matching in G. Then $(V, \mathbf{S})$ is a matroid.*

Proof. Let A and A' be two independent sets with $|A| = |A'| + 1$, and let M and M' be matchings in G which cover the vertices in A and A', respectively. If there exists a vertex $a \in A \setminus A'$ such that M' meets $A' \cup \{a\}$, then condition (2) of Theorem 5.2.1 is trivially satisfied. Otherwise let X be the symmetric difference of M and M'. Then X has to consist of alternating cycles and alternating paths (as in the proof of Theorem 13.2.2): X splits into cycles and paths in which edges of M and M' alternate. As $|A \setminus A'| = |A' \setminus A| + 1$, X has to contain a path P connecting a vertex $x \in A \setminus A'$ to a vertex $y \notin A'$. Then $M' \oplus P$ is a matching meeting $A' \cup \{x\}$. Therefore condition (2) of Theorem 5.2.1 is always satisfied, so that $(V, \mathbf{S})$ is a matroid. □

Given any matroid $(M, \mathbf{S})$ and any subset N of M, the *restriction* $(N, \mathbf{S}|N)$, where $\mathbf{S}|N = \{A \subset N : A \in \mathbf{S}\}$, is again a matroid. Hence Theorem 13.5.1 immediately implies the following result.

Corollary 13.5.2. *Let $G = (V, E)$ be a graph, and let W be a subset of V. Moreover, let $\mathbf{S}$ be the set of all subsets of W consisting of vertices covered by some matching of G. Then $(W, \mathbf{S})$ is a matroid.* □

The matroids described in Corollary 13.5.2 are called *matching matroids*.

Exercise 13.5.3. Derive Theorem 7.3.8 from Corollary 13.5.2.

Exercise 13.5.4. Let $G = (V, E)$ be a graph and A a subset of V. Assume that there exists a matching M covering all vertices in A. Show that there also exists a maximal matching covering A. In particular, each vertex which is not isolated is contained in a maximal matching.

We close this chapter with some remarks. As we have seen, there are efficient algorithms for determining a matching of maximal cardinality. In contrast, determining a non-extendable matching of minimal cardinality is an NP-hard problem – even for planar or bipartite graphs, and even in the case of maximal degree at most 3; see [YaGa80].

The notion of a matching can be generalized as follows. Let $G = (V, E)$ be a graph with $V = \{1, \ldots, n\}$, and let $f : V \to \mathbb{N}_0$ be an arbitrary mapping. A subgraph of G with $\deg v = f(v)$ for $v = 1, \ldots, n$ is called an *f-factor*. Tutte generalized his Theorem 13.1.1 to a necessary and sufficient condition for the existence of an f-factor; see [Tut52]. His general theorem may actually be derived from the 1-factor theorem; see [Tut54]. Anstee [Ans85] gave an algorithmic proof of Tutte's theorem which allows one to determine an f-factor with complexity $O(n^3)$ (or show that no such factor exists). The existence question for f-factors can also be treated in the framework of flow theory, by using the balanced networks already mentioned; see the footnote on page 387. Further generalizations – where the degrees of the vertices are restricted by upper and lower bounds – are considered in [Lov70b] and [Ans85]. A wealth of results concerning matchings as well as an extensive bibliography can be found in the important monograph [LoPl86].

14

Weighted matchings

With matching it's like with playboy bunnies:
You never have it so good.

Jack Edmonds

In the previous chapter, we studied matchings of maximal cardinality (the *cardinality matching problem*). The present chapter is devoted to *weighted* matchings, in particular to the problem of finding a matching of maximal weight in a network (G, w) (the *weighted matching problem*). In the bipartite case, this problem is equivalent to the assignment problem introduced in Example 10.1.4, so that the methods discussed in Chapter 10 apply. Nevertheless, we will give a further algorithm for the bipartite case, the *Hungarian algorithm*, which is one of the best known and most important combinatorial algorithms.

We proceed by explaining the connection between matching problems and the theory of linear programming, even though we generally avoid linear programs in this book. We need this to see the deeper reason why the approach used in the Hungarian algorithm works: its success is due to the particularly simple structure of the corresponding polytope, and ultimately to the total unimodularity of the incidence matrix of a bipartite graph. In this context, the significance of blossoms will become much clearer, as will the reason why the determination of maximal matchings (weighted or not) is considerably more difficult for arbitrary graphs than for bipartite ones. It would make little sense to describe an algorithm for the weighted matching problem in general graphs without using more of the theory of linear programming; for this reason, no such algorithm is presented in this book.

Nevertheless, we will include three interesting applications of weighted matchings: the *Chinese postman problem* (featuring a postman who wants an optimal route for delivering his mail); the determination of shortest paths for the case where edges of negative weight occur; and the decoding of graphical codes. We shall conclude with a few remarks about matching problems with certain additional restrictions – a situation which occurs quite often in practice; we will see that such problems are inherently more difficult.

14.1 The bipartite case

Let $G = (V, E)$ be a bipartite graph with weight function $w : E \to \mathbb{R}$. As usual, the *weight* $w(M)$ of a matching M of G is defined by

$$w(M) \;=\; \sum_{e \in M} w(e).$$

A matching M is called a *maximal weighted matching* if $w(M) \geq w(M')$ holds for every matching M' of G. Obviously, a maximal weighted matching cannot contain any edges of negative weight. Thus such edges are irrelevant in our context, so that we will usually assume w to be nonnegative; but even under this assumption, a maximal weighted matching is not necessarily also a matching of maximal cardinality. Therefore we extend G to a complete bipartite graph by adding all missing edges e with weight $w(e) = 0$; then we may assume that a matching of maximal weight is a complete matching. Similarly, we may also assume $|S| = |T|$ by adding an appropriate number of vertices to the smaller of the two sets (if necessary), and by introducing further edges of weight 0.

In view of the preceding considerations, we will restrict our attention to the problem of determining a perfect matching of maximal weight with respect to a nonnegative weight function w in a complete bipartite graph $K_{n,n}$. We call such a matching an *optimal matching* of $(K_{n,n}, w)$. If we should require a perfect matching of maximal weight in a bipartite graph containing edges of negative weight, we can add a sufficiently large constant to all weights first and thus reduce this case to the case of a nonnegative weight function. Hence we may also treat the problem finding a perfect matching of *minimal* weight, by replacing w by $-w$.

Thus let $w : E \to \mathbb{R}_0^+$ be a weight function for the graph $K_{n,n}$. Suppose the maximal weight of all edges is C. We define the *cost* of a perfect matching M as follows:

$$\gamma(M) \;=\; \sum_{e \in M} \gamma(e),$$

where the cost $\gamma(e)$ of an edge e is given by $\gamma(e) = C - w(e)$. Then the optimal matchings are precisely the perfect matchings of minimal cost. Hence determining an optimal matching in G with respect to the weight function w is equivalent to solving the assignment problem for the matrix $A = (C - w(ij))$ and thus to a special case of the optimal flow problem (compare Examples 10.1.3 and 10.1.4): we just need to find an optimal flow of value n in the flow network described in 10.1.4. As this may be done using the algorithm of Busacker and Gowen, Theorem 10.5.3 implies the following result.

Theorem 14.1.1. *Let w be a nonnegative weight function for $K_{n,n}$. Then an optimal matching can be determined with complexity $O(n^3)$.* □

Exercise 14.1.2. Design a procedure OPTMATCH which realizes the assertion of Theorem 14.1.1.

The complexity $O(n^3)$ given above is, in fact, the best known result for *positive* weight functions on complete bipartite graphs $K_{n,n}$. For non-complete bipartite graphs, one may give a better complexity bound; this follows, for example, from Theorem 10.5.3, using the same approach as above; see also [Tar83, p. 114]. The best known complexity for determining an optimal matching in a general bipartite graph is $O(|V||E|+|V|^2 \log |V|)$, see [FrTa87]; algorithms of complexity $O(|V|^{1/2}|E| \log (|V|C))$ are in [GaTa89] and [OrAh92]. A polynomial version of the network simplex algorithm specialized to the assignment problem can be found in [AhOr92].

14.2 The Hungarian algorithm

In this section, we present a further algorithm for finding an optimal matching in a complete bipartite graph. This algorithm is due to Kuhn [Kuh55, Kuh56] and is based on ideas of König and Egerváry, so that Kuhn named it the *Hungarian algorithm*. Even though his algorithm does not improve on the complexity bound $O(n^3)$ reached in Theorem 14.1.1, it is presented here because it is one of the best-known and (historically) most important combinatorial algorithms.

Thus let $G = (V, E)$ be the complete bipartite graph $K_{n,n}$ with $V = S \dot\cup T$, where $S = \{1, \ldots, n\}$ and $T = \{1', \ldots, n'\}$, and with a nonnegative weight function w described by a matrix $W = (w_{ij})$: the entry w_{ij} is the weight of the edge $\{i, j'\}$. A pair of real vectors $\mathbf{u} = (u_1, \ldots, u_n)$ and $\mathbf{v} = (v_1, \ldots, v_n)$ is called a *feasible node-weighting* if the following condition holds:

$$u_i + v_j \;\geq\; w_{ij} \quad \text{for all } i, j = 1, \ldots, n. \tag{14.1}$$

We will denote the set of all feasible node-weightings $(\mathbf{u}, \mathbf{v})$ by $\mathbf{F}$ and the weight of an optimal matching by D. The following simple result is immediate by summing (14.1) over all edges of the matching M.

Lemma 14.2.1. *For each feasible node-weighting* $(\mathbf{u}, \mathbf{v})$ *and for each perfect matching* M *of* G*, we have*

$$w(M) \;\leq\; D \;\leq\; \sum_{i=1}^{n} (u_i + v_i)\,. \quad \square \tag{14.2}$$

If we can find a feasible node-weighting $(\mathbf{u}, \mathbf{v})$ and a perfect matching M for which equality holds in (14.2), then M has to be optimal. Indeed, it is always possible to achieve equality in (14.2); the Hungarian algorithm will give a constructive proof for this fact.[1] We now characterize the case of equality in (14.2). For a given feasible node-weighting $(\mathbf{u}, \mathbf{v})$, let $H_{\mathbf{u},\mathbf{v}}$ be the subgraph of G with vertex set V whose edges are precisely those ij' for which $u_i + v_j = w_{ij}$ holds; $H_{\mathbf{u},\mathbf{v}}$ is called the *equality subgraph* for $(\mathbf{u}, \mathbf{v})$.

[1] This approach is not a trick appearing out of the blue; we will discuss its theoretical background in the next section.

Lemma 14.2.2. *Let $H = H_{\mathbf{u},\mathbf{v}}$ be the equality subgraph for $(\mathbf{u},\mathbf{v}) \in \mathbf{F}$. Then*

$$\sum_{i=1}^{n} (u_i + v_i) \;=\; D$$

holds if and only if H has a perfect matching. In this case, every perfect matching of H is an optimal matching for (G,w).

Proof. First let $\sum(u_i+v_i) = D$ and suppose that H does not contain a perfect matching. For $J \subset S$, we denote by $\Gamma(J)$ the set of all vertices $j' \in T$ which are adjacent to some vertex $i \in J$ (as usual). By Theorem 7.2.5, there exists a subset J of S with $|\Gamma(J)| < |J|$. (Note that we exchanged the roles of S and T compared to 7.2.5.) Put

$$\delta \;=\; \min\{u_i + v_j - w_{ij} \colon\, i \in J,\, j' \notin \Gamma(J)\}$$

and define $(\mathbf{u}',\mathbf{v}')$ as follows:

$$u'_i = \begin{cases} u_i - \delta & \text{for } i \in J \\ u_i & \text{for } i \notin J \end{cases} \qquad \text{and} \qquad v'_j = \begin{cases} v_j + \delta & \text{for } j' \in \Gamma(J) \\ v_j & \text{for } j' \notin \Gamma(J). \end{cases}$$

Then $(\mathbf{u}',\mathbf{v}')$ is again a feasible node-weighting: the condition $u'_i + v'_j \geq w_{ij}$ might only be violated for $i \in J$ and $j' \notin \Gamma(J)$; but then $\delta \leq u_i + v_j - w_{ij}$, so that $w_{ij} \leq (u_i - \delta) + v_j = u'_i + v'_j$. We now obtain a contradiction:

$$D \leq \sum(u'_i + v'_j) = \sum(u_i + v_j) - \delta|J| + \delta|\Gamma(J)| = D - \delta(|J| - |\Gamma(J)|) < D.$$

Conversely, suppose that H contains a perfect matching M. Then equality holds in (14.1) for each edge of M, and summing (14.1) over all edges of M yields equality in (14.2). This argument also shows that every perfect matching of H is an optimal matching for (G,w). □

The Hungarian algorithm starts with an arbitrary feasible node-weighting $(\mathbf{u},\mathbf{v}) \in \mathbf{F}$; usually, one takes

$$v_1 = \ldots = v_n = 0 \;\text{ and }\; u_i = \max\{w_{ij} : j = 1,\ldots,n\} \;\text{ (for } i = 1,\ldots,n).$$

If the corresponding equality subgraph contains a perfect matching, our problem is solved. Otherwise, the algorithm determines a subset J of S with $|\Gamma(J)| < |J|$ and changes the feasible node-weighting $(\mathbf{u},\mathbf{v})$ in accordance with the proof of Lemma 14.2.2. This decreases the sum $\sum(u_i + v_i)$ and adds at least one new edge ij' with $i \in J$ and $j' \notin \Gamma(J)$ (with respect to $H_{\mathbf{u},\mathbf{v}}$) to the new equality subgraph $H_{\mathbf{u}',\mathbf{v}'}$. This procedure is repeated until the partial matching in H is no longer maximal. Finally, we get a graph H containing a perfect matching M, which is an optimal matching of G as well. For extending the matchings and for changing $(\mathbf{u},\mathbf{v})$, we use an appropriately labelled alternating tree in H. In the following algorithm, we keep a variable δ_j for each $j' \in T$ which may be viewed as a *potential*: δ_j is the current minimal value of $u_i + v_j - w_{ij}$. Moreover, $p(j)$ denotes the first vertex i for which this minimal value is obtained.

Algorithm 14.2.3 (Hungarian algorithm). Let $G = (V, E)$ be a complete bipartite graph with $V = S \dot\cup T$, where $S = \{1, \ldots, n\}$ and $T = \{1', \ldots, n'\}$ and where each edge ij' of G has an associated nonnegative weight w_{ij}. The algorithm determines an optimal matching in G described by an array *mate* (as in Chapter 13). Note that Q denotes a set in what follows (not a queue). Also, we will use a different procedure AUGMENT (compared to Chapter 13), as we are in the bipartite case now.

Procedure HUNGARIAN$(n, w;$ mate$)$

```
(1) for v ∈ V do mate(v) ← 0 od
(2) for i = 1 to n do u_i ← max{w_ij : j = 1, ..., n}; v_i ← 0 od
(3) nrex ← n;
(4) while nrex ≠ 0 do
(5)     for i = 1 to n do m(i) ← false; p(i) ← 0; δ_i ← ∞ od
(6)     aug ← false; Q ← {i ∈ S : mate(i) = 0};
(7)     repeat
(8)         remove an arbitrary vertex i from Q; m(i) ← true; j ← 1;
(9)         while aug = false and j ≤ n do
(10)            if mate(i) ≠ j'
(11)            then if u_i + v_j − w_ij < δ_j
(12)                then δ_j ← u_i + v_j − w_ij; p(j) ← i;
(13)                    if δ_j = 0
(14)                    then if mate(j') = 0
(15)                        then AUGMENT(mate, p, j'; mate);
(16)                            aug ← true; nrex ← nrex −1
(17)                        else Q ← Q ∪ { mate(j')}
(18)                        fi
(19)                    fi
(20)                fi
(21)            fi
(22)            j ← j + 1
(23)        od
(24)        if aug = false and Q = ∅
(25)        then J ← {i ∈ S : m(i) = true}; K ← {j' ∈ T : δ_j = 0};
(26)            δ ← min {δ_j : j' ∈ T \ K};
(27)            for i ∈ J do u_i ← u_i − δ od
(28)            for j' ∈ K do v_j ← v_j + δ od
(29)            for j' ∈ T \ K do δ_j ← δ_j − δ od
(30)            X ← {j' ∈ T \ K : δ_j = 0};
(31)            if mate(j') ≠ 0 for all j' ∈ X
(32)            then for j' ∈ X do Q ← Q ∪ {mate(j')} od
(33)            else choose j' ∈ X with mate(j') = 0;
(34)                AUGMENT(mate, p, j'; mate);
(35)                aug ← true; nrex ← nrex − 1
(36)            fi
```

(37) **fi**
(38) **until** aug = true
(39) **od**

Procedure AUGMENT(mate, p, j'; mate)

(1) **repeat**
(2) $i \leftarrow p(j)$; mate$(j') \leftarrow i$; next $\leftarrow$ mate(i); mate$(i) \leftarrow j'$;
(3) **if** next $\neq 0$ **then** $j' \leftarrow$ next **fi**
(4) **until** next $=0$

Theorem 14.2.4. *Algorithm 14.2.3 determines with complexity $O(n^3)$ an optimal matching for (G, w).*

Proof. Let us call all the operations executed during one iteration of the **while**-loop (4) to (39) a *phase*. First we show by induction on the phases that the array mate always defines a matching in the current equality subgraph $H_{\mathbf{u},\mathbf{v}}$. Because of step (1), this holds after the initialization. Now suppose that our claim holds at the beginning of some phase. During the **repeat**-loop, an *alternating forest* (that is, a disjoint union of alternating trees) B is constructed in $H_{\mathbf{u},\mathbf{v}}$. The outer vertices of this forest are all $i \in S$ satisfying $m(i) =$ true, and the inner vertices of B are the vertices $j' \in T$ with $\delta_j = 0$. If the condition in (14) holds at some point, we have found an augmenting path with end vertex j' in B; then the current matching in $H_{\mathbf{u},\mathbf{v}}$ is replaced by a larger matching using AUGMENT. As B is a subgraph of $H_{\mathbf{u},\mathbf{v}}$, the new matching is again contained in $H_{\mathbf{u},\mathbf{v}}$.

We now turn to the case where the condition in (24) is satisfied, so that we have reached $Q = \emptyset$ without finding an augmenting path in B. If this is the case, subsets $J \subseteq S$ and $K \subseteq T$ are defined in (25) which satisfy $K = \Gamma(J)$. To see this, recall that the vertices in $\Gamma(J)$ are precisely those vertices j' for which $u_i + v_j = w_{ij}$ holds for some $i \in J$. Note that in step (8) all vertices which were an element of Q at some point have been examined, so that (during the **while**-loop (9) to (23)) all vertices $j' \in \Gamma(J)$ were associated with some vertex $i = p(j)$ and δ_j was set to 0. Also, for each vertex j' of K, the vertex mate(j') is contained in J because of (17) and (8). As J contains all the exposed vertices as well (because of (6)), we must have $|\Gamma(J)| < |J|$. Therefore it makes sense to proceed by changing the feasible node-weighting $(\mathbf{u}, \mathbf{v})$ as in the proof of Lemma 14.2.2, and thus to decrease the sum $\sum(u_i + v_i)$. This is done in steps (26) to (28); let us denote the updated vectors (for the moment) by $\mathbf{u}'$ and $\mathbf{v}'$. Now (27) and (28) imply for each edge ij' of $H_{\mathbf{u},\mathbf{v}}$ with $i \in J$ and $j' \in K$

$$u'_i + v'_j \;=\; (u_i - \delta) + (v_j + \delta) \;=\; w_{ij} \quad (\text{as } ij' \in H_{\mathbf{u},\mathbf{v}}),$$

so that all such edges of $H_{\mathbf{u},\mathbf{v}}$ are contained in $H_{\mathbf{u}',\mathbf{v}'}$ as well. Moreover, the condition $u'_i + v'_j \geq w_{ij}$ still holds for all i and j; this is seen as in the proof

of Lemma 14.2.2. When we change $\mathbf{u}$ to $\mathbf{u}'$ in step (27), the potential δ_j also decreases by δ for all $j' \in T \setminus K$; the necessary adjustment is made in step (29). Our definition of δ implies that this process yields at least one $j' \in T \setminus K$ satisfying $\delta_j = 0$. Thus $H_{\mathbf{u}',\mathbf{v}'}$ contains at least one edge ij' with $i \in J$ and $j' \notin K$, that is, an edge leaving J which was not contained in $H_{\mathbf{u},\mathbf{v}}$.[2]

In step (30), all vertices $j' \notin K$ with $\delta_j = 0$ are put into the set X. If there exists an exposed vertex $j' \in X$, we have found an augmenting path in $H_{\mathbf{u}',\mathbf{v}'}$ and the present matching is enlarged using AUGMENT (in steps (33) to (35)). Otherwise, the vertex mate(j') can be added to the alternating forest B for each $j' \in X$, so that Q is no longer empty (step (32)); then the construction of B in the **repeat**-loop continues. Note that the set cardinality of $K = \Gamma(J)$ strictly increases with each execution of steps (25) to (37), so that an exposed vertex has to be reached after having changed $(\mathbf{u}, \mathbf{v})$ at most n times. This shows also that each phase terminates with aug = true and that the matching is extended during each phase.

Obviously, there are exactly n phases. As updating the feasible node-weighting $(\mathbf{u}, \mathbf{v})$ and calling the procedure AUGMENT both need $O(n)$ steps, these parts of a phase contribute at most $O(n^2)$ steps altogether. Note that each vertex is inserted into Q and examined in the inner **while**-loop at most once during each phase. The inner **while**-loop has complexity $O(n)$, so that the algorithm consists of n phases of complexity $O(n^2)$, which yields a total complexity of $O(n^3)$ as asserted. □

Note that each phase of Algorithm 14.2.3 boils down to an application of Algorithm 13.3.2 to the equality subgraph $H_{\mathbf{u},\mathbf{v}}$. Thus the determination of an optimal matching can be reduced to the cardinality matching problem. It is not uncommon that the weighted version of an optimization problem reduces to several applications of the corresponding unweighted problem.

Example 14.2.5. We use Algorithm 14.2.3 for determining an optimal matching of the graph $(K_{5,5}, w)$, where the weight function w is given by the following matrix $W = (w_{ij})$:

$$\begin{pmatrix} 3 & 8 & 9 & 1 & 6 \\ 1 & 4 & 1 & 5 & 5 \\ 7 & 2 & 7 & 9 & 2 \\ 3 & 1 & 6 & 8 & 8 \\ 2 & 6 & 3 & 6 & 2 \end{pmatrix} \quad \begin{matrix} 9 \\ 5 \\ 9 \\ 8 \\ 6 \end{matrix}$$

$$0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \qquad \mathbf{v}\backslash\mathbf{u}$$

[2] In general, $H_{\mathbf{u}',\mathbf{v}'}$ does not contain $H_{\mathbf{u},\mathbf{v}}$ (as we will see in Example 14.2.5): there may be edges ij' with $i \notin J$ and $j' \in K$ which are omitted from $H_{\mathbf{u},\mathbf{v}}$. Fortunately, this does not cause any problems because all vertices $j' \in K$ are saturated by the matching constructed so far; as mentioned above, mate(j') is defined for all $j' \in K$ (and is contained in J). Thus $H_{\mathbf{u}',\mathbf{v}'}$ still contains the current matching.

The numbers on the right-hand side of and below the matrix are the u_i and the v_j, respectively, which the algorithm uses as initial values according to step (2). To make the execution of the algorithm deterministic, we always choose the smallest element of Q in step (8). This gives $i = 1$ in the first phase. We obtain the following values of δ_j and $p(j)$:

$$\begin{array}{ccccccc} 1' & 2' & 3' & 4' & 5' & & j' \\ 6 & 1 & 0 & \infty & \infty & & \delta_j \\ 1 & 1 & 1 & - & - & & p(j). \end{array}$$

The vertex $3'$ is exposed, so that $\{1, 3'\}$ is chosen as the first edge of the matching. During the second phase, we have $i = 2$ and the edge $\{2, 4'\}$ is added to the matching. During the third phase, $Q = \{3, 4, 5\}$; hence $i = 3$ and

$$\begin{array}{ccccccc} 1' & 2' & 3' & 4' & 5' & & j' \\ 2 & 7 & 2 & 0 & 7 & & \delta_j \\ 3 & 3 & 3 & 3 & 3 & & p(j). \end{array}$$

As $4'$ is already saturated, $\text{mate}(4') = 2$ is added to Q. Then $i = 2$ is removed from Q in step (8) and we get

$$\begin{array}{ccccccc} 2 & 1 & 2 & 0 & 0 & & \delta_j \\ 3 & 2 & 3 & 3 & 2 & & p(j). \end{array}$$

Now $5'$ is exposed and AUGMENT yields the new matching consisting of the edges $\{2, 5'\}$, $\{3, 4'\}$ and $\{1, 3'\}$, since we had $p(5) = 2$, $\text{mate}(2) = 4'$, $p(4) = 3$ and $\text{mate}(3) = 0$ before. During the fourth phase, $Q = \{4, 5\}$; then $i = 4$ and

$$\begin{array}{ccccccc} 5 & 7 & 2 & 0 & 0 & & \delta_j \\ 4 & 4 & 4 & 4 & 4 & & p(j). \end{array}$$

As the vertices $4'$ and $5'$ are both saturated, their mates 3 and 2 are inserted into Q. With $i = 2$, $i = 3$, and $i = 5$ in step (8), we obtain the following values for δ_j and $p(j)$:

$$i = 2: \begin{array}{ccccc} 4 & 1 & 2 & 0 & 0 \\ 2 & 2 & 4 & 4 & 4 \end{array}$$

$$i = 3: \begin{array}{ccccc} 2 & 1 & 2 & 0 & 0 \\ 3 & 2 & 4 & 4 & 4 \end{array}$$

$$i = 5: \begin{array}{ccccc} 2 & 0 & 2 & 0 & 0 \\ 3 & 5 & 4 & 4 & 4. \end{array}$$

Now both $2'$ and $p(2) = 5$ are exposed, so that the edge $\{5, 2'\}$ is added to the matching. This ends the fourth phase; up to now, we have found the matching $M = \{\{1, 3'\}, \{2, 5'\}, \{3, 4'\}, \{5, 2'\}\}$ in the equality subgraph $H_{\mathbf{u},\mathbf{v}}$; see Figure 14.1.

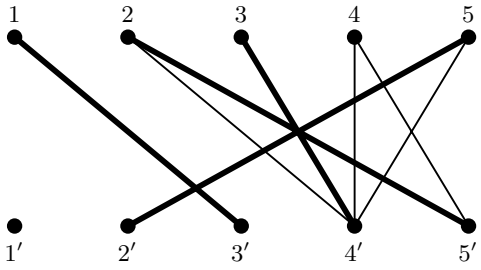


Fig. 14.1. Equality subgraph with a matching

The fifth (and final) phase starts with $Q = \{4\}$; then $i = 4$ and the values of δ_j and $p(j)$ are

$$\begin{array}{ccccc} 5 & 7 & 2 & 0 & 0 \\ 4 & 4 & 4 & 4 & 4. \end{array}$$

Similar to the preceding phase, 2 and 3 are inserted into Q. Then the values of δ_j and $p(j)$ are changed for $i = 2$ and $i = 3$ as follows:

$$i = 2: \quad \begin{array}{ccccc} 4 & 1 & 2 & 0 & 0 \\ 2 & 2 & 4 & 4 & 4 \end{array}$$

$$i = 3: \quad \begin{array}{ccccc} 2 & 1 & 2 & 0 & 0 \\ 3 & 2 & 4 & 4 & 4. \end{array}$$

Now we have reached $Q = \emptyset$ for the first time; thus the feasible node-weighting $(\mathbf{u}, \mathbf{v})$ is changed in accordance with steps (27) and (28). With $J = \{2, 3, 4\}$, $K = \{4', 5'\}$, and $\delta = 1$, we obtain

$$\begin{array}{ccccc} 3 & 8 & 9 & 1 & 6 \\ 1 & 4 & 1 & 5 & 5 \\ 7 & 2 & 7 & 9 & 2 \\ 3 & 1 & 6 & 8 & 8 \\ 2 & 6 & 3 & 6 & 2 \end{array} \quad \begin{array}{c} 9 \\ 4 \\ 8 \\ 7 \\ 6 \end{array}$$

$$\begin{array}{ccccc} 0 & 0 & 0 & 1 & 1 \end{array} \quad \mathbf{v} \backslash \mathbf{u}$$

and the new equality subgraph given in Figure 14.2. Note that the edge $\{5, 4'\}$ was removed from the old equality subgraph, whereas the edge $\{2, 2'\}$ was added. The resulting new equality subgraph is displayed in Figure 14.2. Next the δ_j are updated in step (29) as follows:

$$\begin{array}{ccccc} 1 & 0 & 1 & 0 & 0 \\ 3 & 2 & 4 & 4 & 4. \end{array}$$

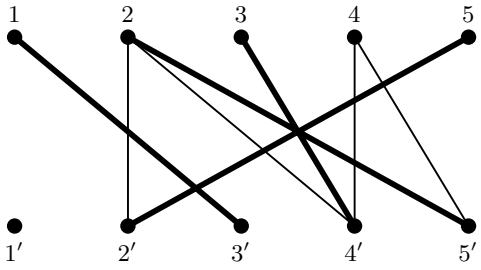


Fig. 14.2. Second equality subgraph

Then $X = \{2'\}$; as $2'$ is not exposed and $\text{mate}(2') = 5$, we insert 5 into Q and get (with $i = 5$):

$$\begin{array}{ccccc} 1 & 0 & 1 & 0 & 0 \\ 3 & 2 & 4 & 4 & 4. \end{array}$$

Again, $Q = \emptyset$. This time the feasible node-weighting $(\mathbf{u}, \mathbf{v})$ is changed as follows (with $J = \{2, 3, 4, 5\}$, $K = \{2', 4', 5'\}$, and $\delta = 1$):

$$\begin{pmatrix} 3 & 8 & \mathbf{9} & 1 & 6 \\ 1 & 4 & 1 & 5 & \mathbf{5} \\ \mathbf{7} & 2 & 7 & 9 & 2 \\ 3 & 1 & 6 & \mathbf{8} & 8 \\ 2 & \mathbf{6} & 3 & 6 & 2 \end{pmatrix} \begin{array}{c} 9 \\ 3 \\ 7 \\ 6 \\ 5 \end{array}$$

$$\begin{array}{ccccc} 0 & 1 & 0 & 2 & 2 \end{array} \qquad \mathbf{v}\backslash\mathbf{u}$$

The new equality subgraph is shown in Figure 14.3: three edges have been added and none removed. The δ_j are then changed to

$$\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 4 & 4 & 4. \end{array}$$

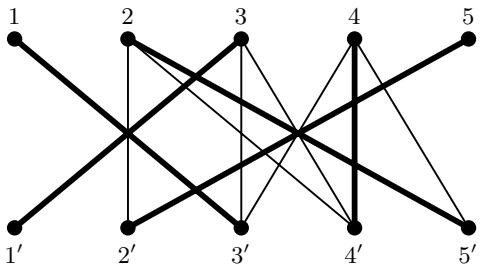


Fig. 14.3. Third equality subgraph with perfect matching

Now $X = \{1', 3'\}$; as $1'$ is exposed, the matching can be enlarged. With $p(1) = 3$, mate$(3) = 4'$, $p(4) = 4$, and mate$(4) = 0$ we obtain the optimal matching $M = \{\{1, 3'\}, \{2, 5'\}, \{3, 1'\}, \{4, 4'\}, \{5, 2'\}\}$, which is displayed in Figure 14.3 and which corresponds to the bold entries in the final matrix above; note that $w(M) = 35$ indeed equals $\sum(u_i + v_i)$ for the feasible node-weighting $(\mathbf{u}, \mathbf{v})$ indicated there.

Exercise 14.2.6. Determine an optimal matching of $K_{9,9}$ with respect to the following weight matrix

$$\begin{pmatrix} 0 & 31 & 24 & 80 & 62 & 39 & 24 & 41 & 42 \\ 31 & 0 & 0 & 34 & 54 & 5 & 51 & 45 & 61 \\ 24 & 0 & 0 & 31 & 32 & 59 & 28 & 44 & 25 \\ 80 & 34 & 31 & 0 & 65 & 45 & 25 & 44 & 47 \\ 62 & 54 & 32 & 65 & 0 & 38 & 48 & 66 & 68 \\ 39 & 5 & 59 & 45 & 38 & 0 & 8 & 25 & 18 \\ 24 & 51 & 28 & 25 & 48 & 8 & 0 & 71 & 66 \\ 41 & 45 & 44 & 44 & 66 & 25 & 71 & 0 & 69 \\ 42 & 61 & 25 & 47 & 68 & 18 & 66 & 69 & 0 \end{pmatrix}$$

Hint: Even though the matrix is quite big, the algorithm works rather fast: the first four phases are almost trivial.

Analyzing Algorithm 14.2.3 again, the reader will realize that the proof actually works for nonnegative weights from an arbitrary ordered abelian group. (This remark seems to be due to Lüneburg.) Recall that an abelian group G is called *ordered* if a partial ordering $\preceq$ is specified which satisfies the following condition:

$$x \preceq y \iff x + z \preceq y + z \qquad \text{for all } x, y, z \in G.$$

Using the ordered group $(\mathbb{R}^+, \cdot)$, we may apply Algorithm 14.2.3 for weights ≥ 1 to determine a matching for which the product of the weights is maximal. More generally, the *algebraic assignment problem* is the problem of finding a perfect matching of maximal (or minimal) weight where the weights come from an ordered commutative monoid; compare Section 3.11. For this problem, we refer to [Zim81] and to [BuHZ77].

Exercise 14.2.7. Show that the bottleneck assignment problem defined in Example 7.4.11 is a special case of the algebraic assignment problem; see [Law76, §5.7] and [GaTa88] for that problem.

Exercise 14.2.8. Determine a *product-optimal* matching for the graph $K_{5,5}$ with respect to the weight matrix of Example 14.2.5; that is, we seek a perfect matching for which the product of the weights of its edges is maximal. Hint: Apply the Hungarian algorithm within the group $(\mathbb{Q}^+, \cdot)$; note that the *zero* of this group is 1, and that the *positive* elements are the numbers ≥ 1.

Exercise 14.2.9. Consider the problem of finding a product-optimal matching of $K_{n,n}$ with respect to a weight matrix all of whose entries are positive integers. Show that this problem is equivalent to determining an optimal matching with respect to some other appropriate weight matrix. Would it be better in practice to use this transformation and then apply Algorithm 14.2.3 directly?

Exercise 14.2.10. Is every optimal matching also product-optimal?

14.3 Matchings, linear programs, and polytopes

The Hungarian algorithm presented in the previous section is an elegant and efficient technique for determining an optimal matching in a weighted bipartite graph. It also allows us to check the correctness of the final result (a feature which is certainly useful when computing smaller examples by hand): we need to check only whether the final vectors $\mathbf{u}$ and $\mathbf{v}$ are indeed a feasible node-weighting and whether the weight of the matching which we have computed is equal to $\sum(u_i + v_i)$; see Lemma 14.2.2.

However, we did not provide any motivation for considering feasible node-weightings in the previous section. It is by no means a coincidence that this approach works and even allows such an easy check of the correctness of the final result. To understand this, we have to appeal to the theory of linear programming, although it is our philosophy to avoid this as far as possible in this book. Nevertheless, linear programming is indispensable for a deeper treatment of combinatorial optimization. Thus we now present a detour into this area; the material dealt with here will be used only rarely in later sections.

A *linear programming problem* (or, for short, an *LP*) is an optimization problem of the following kind: we want to maximize (or minimize, as the case may be) a linear *objective function* with respect to some given *constraints* which have the form of linear equalities or inequalities; note that any equality can be replaced by two inequalities. Formally, one uses the following notation:

$$\begin{aligned} \text{(LP)} \quad & \text{maximize} \quad x_1c_1 + \ldots + x_nc_n \\ & \text{subject to} \quad a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i \quad (i = 1, \ldots, m), \\ & \qquad\qquad x_j \geq 0 \quad (j = 1, \ldots, n). \end{aligned}$$

Sometimes, (some of) the variables x_i are also allowed to be nonrestricted. Writing $A = (a_{ij})$, $\mathbf{x} = (x_1, \ldots, x_n)$, and $\mathbf{c} = (c_1, \ldots, c_n)$, we may write (LP) more concisely:

$$\begin{aligned} \text{(LP}') \quad & \text{maximize} \quad \mathbf{c}\mathbf{x}^T \\ & \text{subject to} \quad A\mathbf{x}^T \leq \mathbf{b}^T \text{ and } \mathbf{x} \geq \mathbf{0}. \end{aligned}$$

For our purposes, A, $\mathbf{b}$, and $\mathbf{c}$ are integral, but we allow $\mathbf{x}$ to have real values. Indeed, the solutions of (LP) are in general not integral but rational. Adding

the condition that $\mathbf{x}$ should be integral to the LP, we get the corresponding *integer linear programming problem* (or, for short, *ILP*).[3] If we restrict $\mathbf{x}$ even further by requiring $x_i \in \{0,1\}$ for $i = 1, \ldots, n$, we have a *zero-one linear program* (or, for short, *ZOLP*). Many of the most important problems of combinatorial optimization can be formulated as a ZOLP; in particular, this holds for optimal matchings.

Example 14.3.1. Let $G = (V, E)$ be a complete bipartite graph with a nonnegative weight function w. Then the optimal matchings of G are precisely the solutions of the following ZOLP:

$$\text{maximize} \quad \sum_{e \in E} w(e) x_e$$

$$\text{subject to } x_e \in \{0,1\} \text{ for all } e \in E \quad \text{and} \quad \sum_{e \in \delta(v)} x_e = 1 \quad \text{for all } v \in V,$$

where $\delta(v)$ denotes the set of edges incident with v. An edge e is contained in the corresponding perfect matching if and only if $x_e = 1$. The constraints above make sure that any solution $\mathbf{x}$ indeed corresponds to a perfect matching. In this particular case, the vectors $\mathbf{x}$ satisfying the constraints (that is, the *admissible* vectors) coincide with the admissible vectors for the corresponding ILP, where the condition $x_e \in \{0,1\}$ is replaced by $x_e \geq 0$. Using the incidence matrix A of G, we can write the ILP concisely as

$$\text{maximize } \mathbf{w}\mathbf{x}^T \text{ subject to } A\mathbf{x}^T = \mathbf{1}^T \text{ and } \mathbf{x} \geq \mathbf{0}, \tag{14.3}$$

where $\mathbf{x} = (x_e)_{e \in E} \in \mathbb{Z}^E$.

Exercise 14.3.2. Describe the problem of finding an optimal integral circulation on a network (G, b, c) as an ILP. Also, describe the problem of finding a maximal spanning tree for a network (G, w) as a ZOLP. Is this an interesting approach to the problem?

[3] Note that the problem SAT treated in Section 2.7 may be viewed as a special case of the problem ILP; see e.g. [PaSt82, Chapter 1]. This implies that ILP is NP-hard, which makes it likely that it cannot be solved in polynomial time. In contrast, LP is a polynomial problem, as the *ellipsoid algorithm* of Khachiyan [Kha79] shows (which is unfortunately of no use for practical purposes); see also [PaSt82, Chapter 7]. A further polynomial algorithm for LP – which is of considerable practical importance – is due to Karmarkar [Kar84]. We refer to [BaJS90] for a nice presentation concerning the complexity of LP, including a detailed description of the algorithm of Karmarkar; the reader will also find further references to the literature there. The original paper of Karmarkar was the starting point for a very large and active area of research. Actually his algorithm is best understood in the context of nonlinear programming; a variation based on a *barrier function* approach is described in [BaSS93, §9.5]. A good discussion of the so-called *path-following methods* can be found in [Gon92] which includes a detailed reference list as well; we also recommend the first part of the monograph [Ter96] on *interior point methods*.

As we wish to apply the theory of linear programming, we have to transform the ILP of Example 14.3.1 into an ordinary LP. Some geometric considerations will be useful here. If the set of all admissible vectors $\mathbf{x} \in \mathbb{R}^n$ for a given LP is bounded and nonempty, then all these vectors form a *polytope*: the convex hull (see Section 7.4) of a finite number of vectors in $\mathbb{R}^n$. It is a fundamental result that optimal solutions for the LP can always be found among the vectors corresponding to *vertices* of the polytope (though there may exist further optimal solutions); here the vertices of the polytope can be defined as those points at which an appropriate objective function achieves its unique maximum over the polytope.

It should now be clear that the incidence vectors of perfect matchings M of G are vertices of the polytope in $\mathbb{R}^E$ defined by the constraints given in Example 14.3.1. Assuming that all the vertices of the polytope actually correspond to perfect matchings, the ZOLP of Example 14.3.1 would be equivalent to the corresponding LP and could be solved – at least in principle – with one of the known algorithms for linear programs.[4] Fortunately, this assumption indeed holds, as the following result of Hoffman and Kruskal [HoKr56] implies; see also [PaSt82, Theorem 13.1].

Result 14.3.3. *Let A be an integer matrix. If A is totally unimodular, then the vertices of the polytope $\{\mathbf{x}\colon A\mathbf{x}^T = \mathbf{b}^T, \mathbf{x} \geq \mathbf{0}\}$ are integral whenever $\mathbf{b}$ is integral.* □

As we assumed G to be bipartite, the incidence matrix A of G is indeed totally unimodular; see Exercise 4.2.13. Hence Result 14.3.3 implies immediately that all vertices of the polytope defined by the LP of Example 14.3.1 are integral, and therefore correspond to perfect matchings of G.

Theorem 14.3.4. *Let A be the incidence matrix of a complete bipartite graph $G = (V, E)$. Then the vertices of the polytope*

$$\mathbf{P} \;=\; \left\{\mathbf{x} \in \mathbb{R}^E\colon\; A\mathbf{x}^T = \mathbf{1}^T, \mathbf{x} \geq \mathbf{0}\right\}$$

coincide with the incidence vectors of perfect matchings of G. Hence the optimal matchings are precisely those solutions of the LP given in Example 14.3.1 which correspond to vertices of $\mathbf{P}$ (for a given weight function). □

Theorem 14.3.4 is certainly interesting, but it does not explain yet why the feasible node-weightings of the previous section work so efficiently. For this purpose, we require also the notion of *duality*. For any linear program

$$\text{(LP)} \qquad \text{maximize } \mathbf{c}\mathbf{x}^T \text{ subject to } A\mathbf{x}^T \leq \mathbf{b}^T \text{ and } \mathbf{x} \geq \mathbf{0},$$

the *dual LP* is the linear program

$$\text{(DP)} \qquad \text{minimize } \mathbf{b}\mathbf{y}^T \text{ subject to } A^T\mathbf{y}^T \geq \mathbf{c}^T \text{ and } \mathbf{y} \geq \mathbf{0},$$

where $\mathbf{y} = (y_1, \ldots, y_m)$. Then the following theorem holds.

[4] We note that this would not be a particularly efficient approach in practice, as the LP under consideration is highly *degenerate*.

Result 14.3.5 (strong duality theorem). *Let* $\mathbf{x}$ *and* $\mathbf{y}$ *be admissible vectors for (LP) and (DP), respectively. Then one has*

$$\mathbf{c}\mathbf{x}^T \;\leq\; \mathbf{b}\mathbf{y}^T$$

with equality if and only if $\mathbf{x}$ *and* $\mathbf{y}$ *are actually optimal solutions for their respective linear programs.* □

Example 14.3.6. Let us return to the situation of Example 14.3.1. As w is nonnegative, we may consider the LP

$$\text{maximize } \mathbf{w}\mathbf{x}^T \text{ subject to } A\mathbf{x}^T \leq \mathbf{1}^T \text{ and } \mathbf{x} \geq \mathbf{0} \tag{14.4}$$

instead of the original LP given there; note that the LP (14.4) likewise yields a polytope with integral vertices; see [PaSt82, Theorem 13.2]. Then the dual linear program is as follows:

$$\text{minimize } \mathbf{1}\mathbf{y}^T \text{ subject to } A^T\mathbf{y}^T \geq \mathbf{w}^T \text{ and } \mathbf{y} \geq \mathbf{0}, \tag{14.5}$$

where $\mathbf{y} = (y_v)_{v\in V}$. In view of $G = K_{n,n}$, it makes sense to use variables $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ corresponding to the partition $V = S \,\dot\cup\, T$ of the vertex set of G instead of the y_v. Then (14.5) becomes

$$\begin{aligned}\text{minimize } \textstyle\sum_{i=1}^n (u_i + v_i) \text{ subject to } & u_i, v_i \geq 0 \quad (i = 1, \ldots, n) \text{ and} \\ & u_i + v_j \geq w_{i,j} \quad (i, j = 1, \ldots, n).\end{aligned} \tag{14.6}$$

Thus the admissible vectors $\mathbf{y}$ for the dual LP (14.5) correspond precisely to the feasible node-weightings $(\mathbf{u}, \mathbf{v})$ in $\mathbb{R}^n \times \mathbb{R}^n$. By Result 14.3.5, an arbitrary perfect matching M of G and an arbitrary feasible node-weighting $(\mathbf{u}, \mathbf{v})$ have to satisfy the condition $\sum(u_i + v_i) \geq w(M)$, and such a perfect matching is optimal if and only if equality holds. This provides us with alternative, more theoretical proofs for the results obtained in Lemma 14.2.1 and Lemma 14.2.2. Thus we have indeed found a deeper reason why the basic idea of the Hungarian algorithm works.

We might now also suspect that the problem of finding optimal matchings in the general case will be considerably harder: the incidence matrix of G will no longer be totally unimodular (see Exercise 4.2.13), so that the linear programming approach cannot possibly work as easily as before. This problem will be addressed in the next section. Note also that network flows and circulations can be treated in a similar way (see Exercise 14.3.2), since the incidence matrix of a digraph is always totally unimodular by Theorem 4.2.5. In particular, Lemma 6.1.2 and Theorem 6.1.6 (max-flow min-cut) may also be derived from Result 14.3.5; see e.g. [PaSt82, Section 6.1].

We hope that the material of this section has convinced the reader that the theory of linear programming is well worth studying, even if one is interested mainly in algorithms concerning graph theoretical problems. Nevertheless,

in my opinion, the first approach to combinatorial optimization should be via graph theory, as this is much more intuitive. We recommend the books [PaSt82], [Chv83], [Schr86], and [NeWo88] for further study.

Let us close this section with some remarks. The Hungarian algorithm shows that we can restrict our attention to feasible node-weightings having integer entries. Again, this is not just a coincidence. There is a simple theoretical explanation: if an integer matrix A is totally unimodular, so is A^T. In particular, another application of Result 14.3.3 shows that the dual program (14.5) for the LP (14.4) of Example 14.3.6 again leads to a polytope with integral vertices. We saw that the Hungarian algorithm simultaneously calculates solutions of the linear program (14.4) and of the dual program (14.5); in fact it can be viewed as a special case of the *primal-dual algorithm* of [DaFF56], which does the same for any linear program (LP) and its dual program (DP); see also [PaSt82]. Moreover, Dijkstra's algorithm, the algorithm of Ford and Fulkerson, and the out-of-kilter algorithm mentioned in Chapter 10 are likewise special cases of the primal-dual algorithm.

Let us also state a result which shows that the vertices of a polytope are integral even under weaker conditions than the total unimodularity of the matrix A; see [Hof74] and [EdGi77].

Result 14.3.7 (total dual integrality theorem). *If the dual program (DP) admits an optimal integral solution for every choice of the objective function* $\mathbf{c}$ *of (LP), then the polytope*

$$\mathbf{P} \;=\; \left\{\mathbf{x} \in \mathbb{R}^n \colon\; A\mathbf{x}^T \leq \mathbf{b}^T,\, \mathbf{x}^T \geq \mathbf{0}\right\}$$

has integral vertices. □

Linear programs having the property described in Result 14.3.7 are called *totally dual integral*.

Finally, we give a few more references. On the one hand, we recommend four interesting surveys which treat the questions considered in this section more thoroughly: [Hof79] for the role of unimodularity in combinatorial applications of linear inequalities; [Lov79] about integral programs in graph theory; and [EdGi84] and [Schr84] about total dual integrality. On the other hand (and on a much deeper level), the reader may find an encyclopedic treatment of the polyhedral approach to combinatorial optimization in [Schr03].

14.4 The general case

In this section, we will discuss optimal matchings in arbitrary graphs and the corresponding linear programs without giving any proofs. Let $G = (V, E)$ be a complete graph K_{2n} with a nonnegative weight function w.[5] As in Example 14.3.1, the optimal matchings of (G, w) are precisely the solutions of the

[5] Using arguments similar to those employed in Section 14.1, one sees that determining a matching of maximal weight – as well as determining a perfect matching

integer linear program

$$\text{maximize } \mathbf{w}\mathbf{x}^T \text{ subject to } A\mathbf{x}^T = \mathbf{1}^T \text{ and } \mathbf{x} \geq \mathbf{0}, \tag{14.7}$$

where $\mathbf{x} = (x_e)_{e \in E}$ and where A is the incidence matrix of G. Unfortunately, by Exercise 4.2.13, A is not totally unimodular in general, so that the methods used in the previous section cannot be transferred immediately. Indeed, the linear program corresponding to (14.7) usually admits rational solutions: the corresponding polytope may have vertices which are not integral. The following simple example for this phenomenon is taken from [Edm67a].

Example 14.4.1. Consider the graph $G = K_6$ with the weights shown in Figure 14.4; edges which are missing in this figure have weight 0. It is easy to check that the bold edges form an optimal matching M, which has weight $w(M) = 18$. On the other hand, the rational values for x_e shown in Figure 14.5 lead to a better value for the objective function: $\mathbf{w}\mathbf{x}^T = 19$; incidentally, this is the optimal solution for the corresponding linear program.

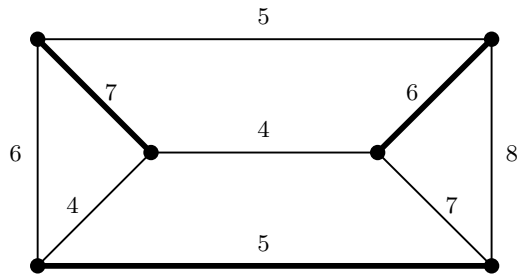


Fig. 14.4. An optimal matching

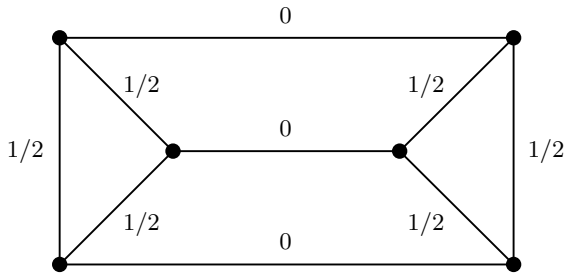


Fig. 14.5. A better rational solution

of maximal weight, or a perfect matching of minimal weight – in a non-bipartite graph reduces to the problem of determining an optimal matching in a complete graph K_{2n}.

One possible way to avoid this unpleasant situation would be to *cut off* the non-integral vertices of the polytope $\mathbf{P'} = \{\mathbf{x}\colon A\mathbf{x}^T = \mathbf{1}^T, \mathbf{x} \geq \mathbf{0}\}$ by adding further inequalities; this approach is quite common in integer linear programming (*cutting plane algorithms*); see [PaSt82, Chapter 14].

Thus we add appropriate inequalities to the LP (14.7) until the enlarged system of linear inequalities corresponds to a polytope which is the convex hull of the incidence vectors of the perfect matchings of G (that is, of the solutions of the ILP (14.7) for an appropriate function w). The following result due to Edmonds [Edm65a] makes this more precise.

Result 14.4.2. *Let $G = (V, E)$ be a graph with an even number of vertices. Then the polytope defined by (14.7) together with the system of additional linear inequalities*

$$\sum_{e \in E|S} x_e \leq \frac{|S| - 1}{2} \quad \textit{for each subset } S \textit{ of } V \textit{ of odd cardinality} \tag{14.8}$$

is the convex hull $P = P(G)$ of the incidence vectors of the perfect matchings of G. □

It is clear that the incidence vectors of perfect matchings satisfy (14.8). The interesting part of Result 14.4.2 states that any vector in $\mathbb{R}^E$ which satisfies both (14.7) and (14.8) necessarily is a convex combination of perfect matchings.

Corollary 14.4.3. *Let G be a complete graph K_{2n} with incidence matrix A. Given any nonnegative weight function w, the linear program*

$$\textit{maximize } \mathbf{w}\mathbf{x}^T \textit{ subject to (14.7) and (14.8)}$$

has an optimal integral solution, which is the incidence vector $\mathbf{x}$ of an optimal matching of (G, w). □

Edmonds' proof for Result 14.4.2 is constructive – that is, algorithmic. Result 14.4.2 can also be derived by using the following result of Cunningham and Marsh [CuMa78] together with Result 14.3.7.

Result 14.4.4. *Let $G = (V, E)$ be a graph with incidence matrix A. Then the system of inequalities*

$$A\mathbf{x}^T \leq \mathbf{1}^T, \quad \mathbf{x} \geq \mathbf{0} \quad \textit{and}$$

$$\sum_{e \in E|S} x_e \leq \tfrac{|S|-1}{2} \quad \textit{for each subset } S \textit{ of } V \textit{ of odd cardinality}$$

is totally dual integral. □

Again, the original proof for this result was algorithmic. Short combinatorial proofs of Results 14.4.2 and 14.4.4 can be found in [Schr83a, Schr83b].

It should be clear that one may now proceed in analogy with the bipartite case: replace the ILP (14.7) by the LP in Corollary 14.4.3 and apply an appropriate special version of the primal-dual algorithm for solving it. In fact, the algorithms most frequently used in practice use this approach. In particular, this holds for the first solution of the ILP (14.7), which was given by Edmonds [Edm65a]; his method has a complexity of $O(n^4)$, but this can be improved to $O(n^3)$; see [Gab76] and [Law76]. A different algorithm with complexity $O(n^3)$ is in [CuMa78].

The fastest algorithms known for determining a matching of maximal weight in an arbitrary graph have complexity $O(|V||E|+|V|^2 \log |V|)$, as in the bipartite case; see [Gab90]. A further fast algorithm (which takes the maximal size of the weights into account) is due to [GaTa91]. An algorithm treating the interesting special case where the weight function on a graph K_{2n} is given by the distances between $2n$ points in the Euclidean plane can be found in [Vai89]; this algorithm has a complexity of $O(n^{5/2}(\log n)^4)$.

All the algorithms for determining an optimal matching in K_{2n} mentioned above are considerably more involved than corresponding algorithms for the bipartite case. This is not surprising if we consider the additional inequalities needed in (14.8) for subsets of odd cardinality; note that these correspond to the fact that blossoms may occur. As it seems almost impossible to give sufficient motivation for an algorithm which does not explicitly use the methods of linear programming, we decided not to treat any algorithm for the determination of optimal matchings in arbitrary graphs. Hence we just state the following result for later use.

Result 14.4.5. *It is possible to determine with complexity $O(n^3)$ an optimal matching in K_{2n} with respect to a given nonnegative weight function w.* □

For a proof of Result 14.4.5, we refer to [Law76] or [BaDe83]. In [PaSt82], an algorithm with complexity $O(n^4)$ is derived from the primal-dual algorithm. A method which avoids the explicit use of linear programming – and which is, not surprisingly, less motivated – can be found in [GoMi84]. Finally, we also recommend the monograph [Der88].

We close this section with some remarks. The inequalities in Result 14.4.4 define the *matching polytope* $\mathbf{M}(G)$ of the graph G, whereas those in Corollary 14.4.3 describe the *perfect matching polytope* $\mathbf{P}(G)$. These two polytopes are the convex hulls of the incidence vectors of the matchings and the perfect matchings of G, respectively. One might also wonder what the *linear* span of the associated vectors in $\mathbb{R}^E$ is. This question is trivial for $\mathbf{M}(G)$: then any edge forms a matching in G by itself, so that the linear span is all of $\mathbb{R}^E$. However, the problem becomes interesting (and quite difficult) for $\mathbf{P}(G)$; a solution can be found in [EdLP82]. Lovász [Lov85] asked the related question about the *lattice* generated by the incidence vectors of the perfect matchings in $\mathbb{Z}^E$ (that is, the set of integral linear combinations of these vectors) and

derived interesting partial results. Let us pose two exercises regarding this problem.

Exercise 14.4.6. Extend Corollary 7.2.7 to regular bipartite multigraphs.

Exercise 14.4.7. Let G be a bipartite graph, and let $\mathbf{L}(G)$ be the lattice in $\mathbb{Z}^E$ generated by the incidence vectors of the perfect matchings of G, and $\mathbf{H}(G)$ the linear span of $\mathbf{L}(G)$ in $\mathbb{R}^E$. Show that $\mathbf{L}(G) = \mathbf{H}(G) \cap \mathbb{Z}^E$ [Lov85]. Hint: Use Exercise 14.4.6.

The result of Exercise 14.4.7 does not extend to arbitrary graphs, as shown by [Lov85]: the Petersen graph provides a counterexample. The general case is treated in [Lov87]. Related problems can be found in [JuLe88, JuLe89] and [Rie91], where lattices corresponding to the 2-matchings of a graph and lattices corresponding to the bases of a matroid are examined.

For some practical applications in which n is very large even algorithms for determining an optimal matching with complexity $O(n^3)$ are not fast enough; in this case, one usually resorts to approximation techniques. In general, these techniques will not find an optimal solution but just a reasonable approximation; to make up for this, they have the advantage of being much faster. We refer the interested reader to [Avi78, Avi83] and to [GrKa88]. Two alternatives to using heuristics for large values of n are either to use appropriate LP-relaxations to determine minimal perfect matchings on suitable sparse subgraphs, or to use post-optimization methods. We refer to [GrHo85] and to [DeMe91]; one of the best practical methods at present seems to be the one given in [ApCo93].

14.5 The Chinese postman

This section is devoted to an interesting application of optimal matchings in K_{2n}. The following problem due to Kwan [Kwa62] concerns a postman who has to deliver the mail for a (connected) system of streets: our postman wants to minimize the total distance he has to walk by setting up his tour suitably. This problem is nowadays generally known as the *Chinese postman problem*.

Problem 14.5.1 (Chinese postman problem, CPP). Let $G = (V, E)$ be a connected graph, and let $w\colon E \to \mathbb{R}_0^+$ be a length function on G. We want to find a closed walk C of minimal length $w(C)$ which contains each edge of G at least once.[6]

[6] Note that we view the edges of our graph as (segments of) streets here, and the vertices as intersections (or dead ends), so that each edge certainly needs to be traversed to deal with the houses in this street; in this rather simplistic model we neglect the need for having to cross the street to deliver the mail to houses on opposite sides. Hence it might be more realistic to consider the directed case and use the complete orientation of G; see Exercise 14.5.6.

If G should be Eulerian, the solution of the CPP is trivial: any Euler tour C will do the job. Recall that G is Eulerian if and only if each vertex of G has even degree (Theorem 1.3.1) and that an Euler tour C can then be constructed with complexity $O(|E|)$ (Example 2.5.2).

If G is not Eulerian, we use the following approach. Let X be the set of all vertices of G with odd degree. We add a set E' of edges to G such that the following three conditions are satisfied:

(a) Each edge $e' \in E'$ is parallel to some edge $e \in E$; we extend w to E' by putting $w(e') = w(e)$.
(b) In (V, E'), precisely the vertices of X have odd degree.
(c) $w(E')$ is minimal: $w(E') \leq w(E'')$ for every set E'' satisfying (a) and (b).

Then $(V, E \,\dot\cup\, E')$ is an Eulerian multigraph, and any Euler tour induces a closed walk of minimal length $w(E)+w(E')$ in G. It is rather obvious that any solution of CPP can be described in this way. We now state – quite informally – the algorithm of Edmonds and Johnson [EdJo73] for solving the CPP. Note that $|X|$ is even by Lemma 1.1.1.

Algorithm 14.5.2. Let $G = (V, E)$ be a connected graph with a length function $w\colon E \to \mathbb{R}_0^+$.

Procedure CPP$(G, w; C)$

(1) $X \leftarrow \{v \in V\colon \deg v \text{ is odd}\}$;
(2) Determine $d(x, y)$ for all $x, y \in X$.
(3) Let H be the complete graph on X with weight function $d(x, y)$. Determine a perfect matching M of minimal weight for (H, d).
(4) Determine a shortest path W_{xy} from x to y in G and, for each edge in W_{xy}, add a parallel edge to G (for all $xy \in M$). Let G' be the multigraph thus defined.
(5) Determine an Euler tour C' in G' and replace each edge of C' which is not contained in G by the corresponding parallel edge in G. Let C be the closed walk in G arising from this construction.

Step (2) can be performed using Algorithm 3.9.1; however, if $|X|$ is small, it might be better to run Dijkstra's algorithm several times. Determining shortest paths explicitly in step (4) can be done easily by appropriate modifications of the algorithms already mentioned; see Exercise 3.9.3 and 3.7.3. In the worst case, steps (2) and (4) need a complexity of $O(|V|^3)$. Step (3) can be executed with complexity $O(|X|^3)$ by Result 14.4.5; note that determining a perfect matching of minimal weight is equivalent to determining an optimal matching for a suitable auxiliary weight function; see Section 14.1. Finally, step (5) has complexity $O(|E'|)$ by Example 2.5.2. Thus we get a total complexity of $O(|V|^3)$.

It still remains to show that the algorithm is correct. Obviously, the construction in step (4) adds, for any matching M of H, a set E' of edges to G which satisfies conditions (a) and (b) above; the closed walk in G arising from

this construction has length $w(E) + d(M)$, where $d(M)$ is the weight of M with respect to d. Thus it is reasonable to choose a matching M of minimal weight in step (3). However, it is not immediately clear that there cannot be some other set E' of edges leading to a solution of even smaller weight. We need the following lemma.

Lemma 14.5.3. *Let $G = (V, E)$ be a connected graph with length function $w : E \to \mathbb{R}_0^+$. Moreover, let H be the complete graph on a subset X of V of even cardinality; the edges of H are assigned weight $d(x, y)$, where d denotes the distance function in G with respect to w. Then, for each perfect matching M of H with minimal weight and for each subset E_0 of E for which any two vertices of X have the same distance in G and in (V, E_0), the inequality $d(M) \leq w(E_0)$ holds.*

Proof. Let $M = \{x_1y_1, \ldots, x_ny_n\}$ be a perfect matching with minimal weight in H. Then $d(M) = d(x_1, y_1) + \ldots + d(x_n, y_n)$. Moreover, let P_i be a shortest path from x_i to y_i in (V, E_0) (for $i = 1, \ldots, n$). By hypothesis, $w(P_i) = d(x_i, y_i)$. We claim that no edge e with $w(e) \neq 0$ can be contained in more than one of the paths P_i; if we prove this claim, the assertion of the lemma follows. Suppose our claim is wrong. Then we may assume

$$P_1 = x_1 \overset{P_1'}{\text{---}} u \overset{e}{\text{---}} v \overset{P_1''}{\text{---}} y_1 \quad \text{and} \quad P_2 = x_2 \overset{P_2'}{\text{---}} u \overset{e}{\text{---}} v \overset{P_2''}{\text{---}} y_2,$$

which implies

$$\begin{aligned} d(x_1, y_1) + d(x_2, y_2) &= d(x_1, u) + w(e) + d(v, y_1) + d(x_2, u) + w(e) + d(v, y_2) \\ &> d(x_1, u) + d(u, x_2) + d(y_1, v) + d(v, y_2) \\ &\geq d(x_1, x_2) + d(y_1, y_2). \end{aligned}$$

But then replacing x_1y_1 and x_2y_2 by x_1x_2 and y_1y_2 in M would yield a perfect matching of smaller weight, a contradiction. □

Theorem 14.5.4. *Algorithm 14.5.2 calculates with complexity $O(|V|^3)$ a solution of the CPP.*

Proof. We already know that Algorithm 14.5.2 yields a closed walk of length $w(E) + d(M)$ containing each edge of G, where $d(M)$ is the minimal weight of a perfect matching of (H, d).

Now suppose that E' is an arbitrary set of edges satisfying conditions (a) to (c). Then E' induces a closed walk of weight $w(E) + w(E')$ which contains all edges of G. We have to show $w(E') \geq d(M)$. Suppose Z is a connected component of (V, E') containing at least two vertices. Then we must have $Z \cap X \neq \emptyset$: otherwise, we could omit all edges of E' which are contained in Z and the remaining set of edges would still satisfy (a) and (b). As X is the set of vertices of (V, E') with odd degree, $|Z \cap X|$ has to be even by Lemma 1.1.1. Thus the connected components of (V, E') induce a partition $X_1, \ldots, X_k$ of

X into sets of even cardinality so that any two vertices in X_i are connected by a path in E'.

Let $x, y \in X_i$, and let P_{xy} be the path from x to y in E'. Then P_{xy} must be a shortest path from x to y in G: otherwise, the edges of P_{xy} could be replaced by the edges of a shortest path from x to y, which would yield a set E'' of edges satisfying (a) and (b) and $w(E'') < w(E')$. Now, trivially, P_{xy} is also a shortest path from x to y in (V, E'). Denote the connected component of (V, E') corresponding to X_i by Z_i, and let E'_i be the set of edges of E' which have both end vertices in Z_i. Moreover, let H_i be the complete graph on Z_i with weights $d(x, y)$ (where d is the distance function in G or in (Z_i, E'_i)). Then Lemma 14.5.3 yields $d(M_i) \leq w(E'_i)$ for each perfect matching M_i of minimal weight in H_i. Obviously, $M_1 \cup \ldots \cup M_k$ is a perfect matching of H, and $E' = E'_1 \cup \ldots \cup E'_k$. Hence we obtain the desired inequality

$$w(E') \;=\; w(E'_1) + \ldots + w(E'_k) \;\geq\; d(M_1) + \ldots + d(M_k) \;\geq\; d(M). \quad \square$$

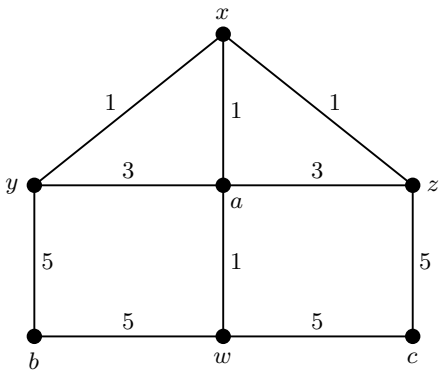


Fig. 14.6. A graph

Example 14.5.5. Let G be the graph displayed in Figure 14.6. Then $X = \{x, y, z, w\}$, so that we get the complete graph H shown in Figure 14.7. The edges xw and yz form a perfect matching of minimal weight of H; the corresponding paths are (x, a, w) and (y, x, z). Hence we replace the corresponding edges in G by two parallel edges each; this yields the multigraph G' in Figure 14.8. Now it is easy to find an Euler tour in G', for example $(x, y, b, w, c, z, x, y, a, x, a, w, a, z, x)$ with length $30 + 4 = 34$.

Exercise 14.5.6. We consider the directed version of the CPP: let G be a digraph with a nonnegative length function w; we want to find a directed closed walk of minimal length containing each edge of G at least once. Hint: Reduce this problem to the problem of determining an optimal circulation [EdJo73].

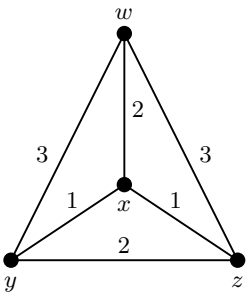


Fig. 14.7. The complete graph H

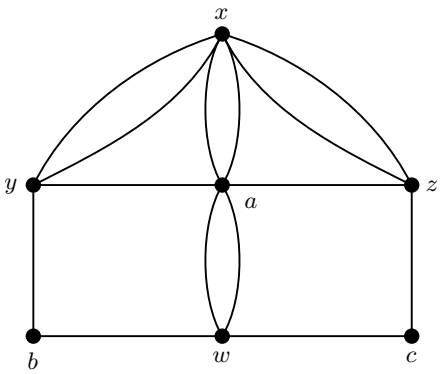


Fig. 14.8. The corresponding Eulerian multigraph

Theorem 14.5.4 and Exercise 14.5.6 (together with a corresponding algorithm for determining an optimal circulation) show that there are good algorithms for the CPP for directed graphs as well as for undirected graphs. In contrast, the CPP for mixed graphs is NP-complete, so that most likely there is no polynomial solution; see [Pap76] or [GaJo79]. A cutting plane algorithm for the mixed CCP is in [NoPi96], and some applications of the CPP are discussed in [Bar90].

14.6 Matchings and shortest paths

This section deals with applications of matchings to the problem of determining shortest paths in a network on an *undirected* graph without cycles of negative length. We remind the reader that our usual transformation to the directed case – replacing a graph G by its complete orientation – will not work in this situation, because an edge $e = \{u, v\}$ of negative weight $w(e)$ in (G, w) would yield a directed cycle $u \text{ — } v \text{ — } u$ of negative length $2w(e)$ in $(\vec{G}, w)$,

whereas all the algorithms given in Chapter 3 apply only to graphs without such cycles. We describe a solution for this path problem below; it is due to Edmonds [Edm67a].

The first step consists of transforming the given problem to the problem of determining an f-factor in an appropriate auxiliary graph; this problem was already mentioned at the end of Section 13.5. In our case, the only values $f(v)$ will take are 1 and 2; however, the auxiliary graph might contain loops. Note that a loop $\{v, v\}$ adds 2 to the degree $\deg v$ of a vertex v. In what follows, we call a path from s to t an *$\{s,t\}$-path.*

Lemma 14.6.1. *Let $N = (G, w)$ be a network on a graph $G = (V, E)$ with respect to a weight function $w\colon E \to \mathbb{R}$, and assume that there are no cycles of negative length in N. Let s and t be two vertices of G, and let G' be the graph which results from adding the loop $\{v, v\}$ to G for each vertex $v \neq s, t$. Extend the weight function w to G' by putting $w(v, v) = 0$. Then each $\{s,t\}$-path P in G may be associated with an f-factor $F = F(P)$ in G', where f is given by*

$$f(s) = f(t) = 1 \quad \textit{and} \quad f(v) = 2 \textit{ for all } v \neq s, t, \tag{14.9}$$

so that the weight of P always equals that of the corresponding f-factor F. Moreover, the problem of determining a shortest $\{s,t\}$-path in (G, w) is equivalent to determining a minimal f-factor in (G', w).

Proof. Given an $\{s,t\}$-path P in G, put

$$F \;=\; P \cup \{\{v, v\} : v \text{ is not contained in } P\}\,.$$

Obviously, F is an f-factor for G', as the loop $\{v, v\}$ increases the degree of v in F to 2 whenever v is not contained in P. By our definition of w for loops, $w(F) = w(P)$.

Conversely, let F be an f-factor for G'; we want to construct an $\{s,t\}$-path P from F. As s has degree 1 in F, there is exactly one edge sv_1 in F. Now v_1 has degree 2 in F, so that there exists precisely one further edge in F incident with v_1, say v_1v_2; note that this edge cannot be a loop. Continuing in this manner, we construct the edge sequence of a path P with start vertex s in G. As the only other vertex of degree 1 in F is t, t must be the end vertex of P.

Note that it is quite possible that there are not only loops among the remaining edges of F: these edges might contain one or more cycles. In other words, in general we will have $F \neq F(P)$, so that the correspondence given above is not a bijection. However, our assumption that there are no cycles of negative length in (G, w) guarantees at least $w(P) \leq w(F)$, which proves the final assertion. □

Next we show how one may reduce the determination of a minimal f-factor for the special case where $f(v) \in \{1, 2\}$ to the determination of a minimal perfect matching in an appropriate auxiliary graph whose size is polynomial in the size of the original graph. As already mentioned in Section 13.5, the

general existence problem for arbitrary f-factors can be reduced to the general existence problem for perfect matchings; see [Tut54].

Lemma 14.6.2. *Let $G = (V, E)$ be a graph (where loops are allowed), and let $f : V \to \mathbb{N}$ be a function with $f(v) \in \{1, 2\}$ for all $v \in V$. Then the f-factors of G correspond to perfect matchings of a suitable auxiliary graph H with at most $5|E|$ edges and at most $2|V| + 2|E|$ vertices. If there also is a weight function $w : E \to \mathbb{R}$ on G given, a weight function w on H can be defined in such a way that the weight $w(F)$ of an f-factor F is always equal to the weight $w(M)$ of the corresponding perfect matching M.*

Proof. Our transformation consists of two steps. First, the given f-factor problem for G is transformed to an equivalent problem for an auxiliary graph H' for which each non-loop edge is incident with at least one vertex v satisfying $f(v) = 1$. Thus let $e = uv \in E$ be an edge with $u \neq v$ and $f(u) = f(v) = 2$. We subdivide e by introducing two new vertices u_e, v_e; replace the edge e by the path

$$P_e : \; u \text{ —— } u_e \text{ —— } v_e \text{ —— } v;$$

and extend f by putting $f(u_e) = f(v_e) = 1$. By performing this operation for all edges $e = uv$ with $f(u) = f(v) = 2$ and $u \neq v$, we obtain the desired graph H'. Now let F be an f-factor in G. Then F yields an f-factor F' in H' as follows: we replace each edge $e = uv \in F$ with $f(u) = f(v) = 2$ and $u \neq v$ by the edges uu_e and vv_e; moreover, we add for each edge $e = uv$ with $f(u) = f(v) = 2$ and $u \neq v$ which is not in F the edge $u_e v_e$ to F'. Under this operation, each f-factor in H' actually corresponds to an f-factor in G. We can also make sure that the weights of corresponding f-factors F and F' are equal: for each edge $e = uv$ with $f(u) = f(v) = 2$ and $u \neq v$, we define the weights of the edges on P_e as

$$w(uu_e) \;=\; w(vv_e) \;=\; \frac{w(e)}{2} \quad \text{and} \quad w(u_e v_e) \;=\; 0.$$

In the second step of the transformation, we define a graph H which results from H' by splitting each vertex v with $f(v) = 2$ into two vertices:[7] we replace v by two vertices v' and v''; we replace each edge $e = uv$ with $u \neq v$ by two edges $e' = uv'$ and $e'' = uv''$; finally, each loop $\{v, v\}$ with $f(v) = 2$ is replaced by the edge $v'v''$. These operations are well-defined because of the transformations performed in the first step: H' does not contain any edges $e = uv$ with $f(u) = f(v) = 2$ and $u \neq v$. Let us denote the resulting graph by H.

It is now easy to see that the f-factors F' of H' correspond to the perfect matchings M of H. Note that at most one of the two parts of a split edge $e = uv$ (with $f(v) = 2$) can be contained in a perfect matching M of H, since

[7] Note that this condition can hold only for *old* vertices, that is, vertices which were contained in G.

we must have $f(u) = 1$ in that case. Note that this correspondence between f-factors and perfect matchings is, in general, not bijective: if F' contains two edges $e_1 = u_1v$ and $e_2 = u_2v$ (where $f(v) = 2$ and $f(u_1) = f(u_2) = 1$), M might contain either u_1v' and u_2v'' or u_1v'' and u_2v'. Thus, in general, there are several perfect matchings of H which correspond to the same f-factor of H'. However, the weights of corresponding f-factors and perfect matchings agree if we put

$$w(e') \;=\; w(e'') \;=\; w(e)$$

for split edges e' and e''. □

By performing the transformations of Lemmas 14.6.1 and 14.6.2 successively, we obtain the desired reduction of the determination of a shortest path between two vertices s and t in an undirected network (G, w) without cycles of negative length to the determination of a perfect matching of minimal weight in an appropriate auxiliary graph H (with respect to a suitable weight function). As the number of vertices of H is linear in the number of edges of G, Result 14.4.5 yields the following conclusion.

Theorem 14.6.3. *Let $N = (G, w)$ be a network on a graph $G = (V, E)$, where $w\colon E \to \mathbb{R}$, and let s and t be two vertices of G. If N does not contain cycles of negative length, one may determine with complexity $O(|E|^3)$ a shortest path from s to t.* □

Example 14.6.4. Consider the network (G, w) given in Figure 14.9. The bold edges form a path

$$P:\; s \text{ —— } c \text{ —— } b \text{ —— } t$$

of length $w(P) = 0$, which corresponds to the f-factor

$$F \;=\; \{\{a, a\}, sc, cb, bt\}$$

of weight $w(F) = 0$ in the graph G' shown in Figure 14.10, where $f(a) = f(b) = f(c) = 2$ and $f(s) = f(t) = 1$. Again, F consists of the bold edges.

Now we perform the transformations of Lemma 14.6.2. First, when H' is constructed, the edges $e = ab$ and $g = bc$ are divided into paths of length 3. We obtain the auxiliary graph H' with the f-factor

$$F' \;=\; \{\{a, a\}, sc, cc_g, bb_g, bt, a_eb_e\}$$

corresponding to F, where $f(a) = f(b) = f(c) = 2$ and $f(v) = 1$ for all other vertices v. Note that F' indeed has weight $w(F') = 0$. Figure 14.11 shows H' and F'; as usual, F' consists of the bold edges.

Finally, in the second step of the transformation, the three vertices a, b, c with $f(a) = f(b) = f(c) = 2$ are divided into two vertices each. This yields the graph H shown in Figure 14.12 and the perfect matching

$$K \;=\; \{aa', sc', c''c_g, b''b_g, b't, a_eb_e\}$$

of weight $w(K) = 0$ corresponding to the f-factor F'.

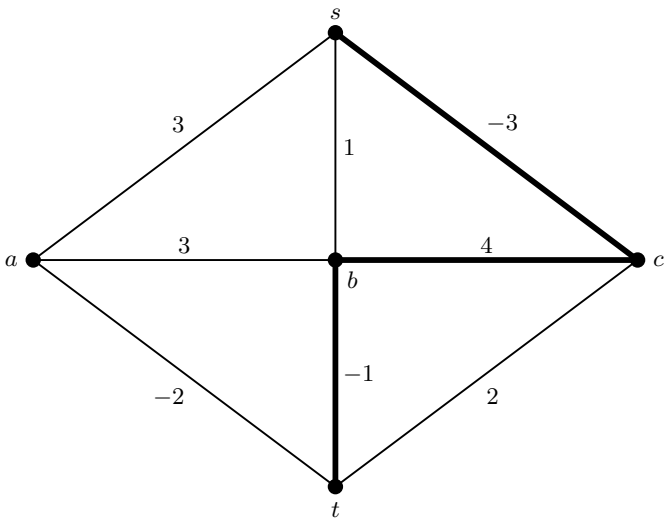


Fig. 14.9. A path in G

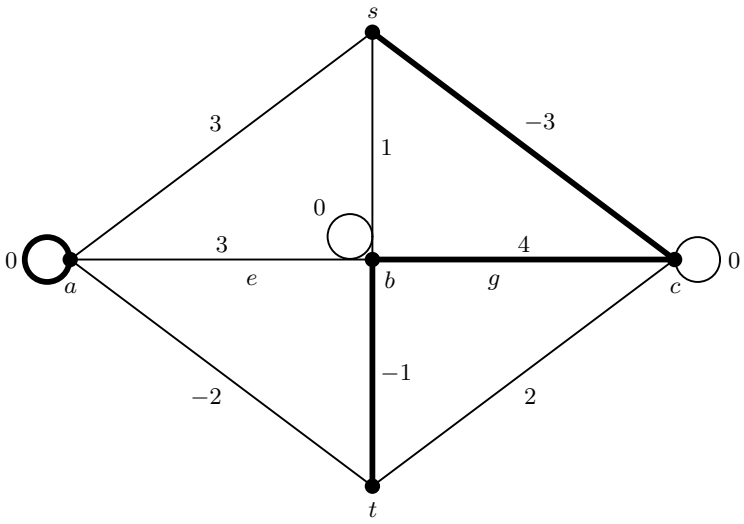


Fig. 14.10. The corresponding f-factor in G'

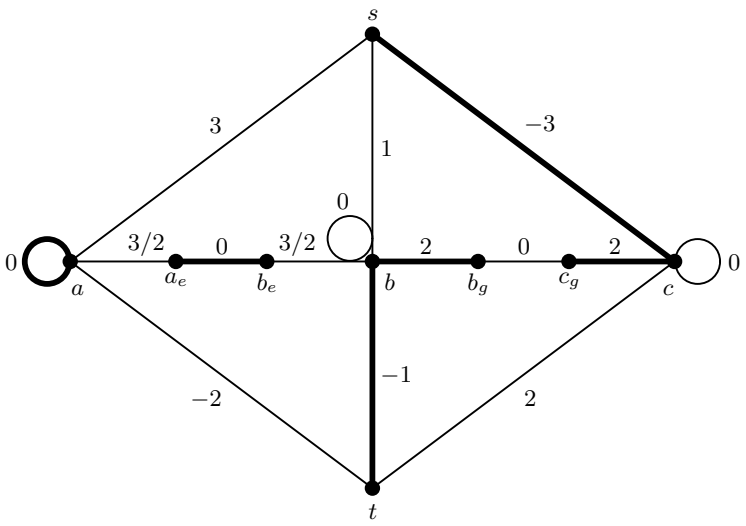


Fig. 14.11. The corresponding f-factor in H'

Exercise 14.6.5. Determine an $\{s,t\}$-path of shortest length as well as the corresponding f-factors and a corresponding perfect matching of minimal weight for the network of Example 14.6.4.

Exercise 14.6.6. Discuss the transformation method given above for the case in which (G,w) contains cycles of negative length. What will go wrong then?

Now consider a network (G,w) on a digraph G which does not contain directed cycles of negative length. Then the problem of determining a shortest directed path from s to t can be transformed to the problem of determining a perfect matching of minimal weight in a bipartite graph – that is, to the assignment problem; see [HoMa64] and also [AhMO93, Chapter 12.7]. As we have already seen two efficient algorithms for determining shortest paths for this case in Chapter 3, we will not present this transformation here. In practice, the reverse approach is more common: the assignment problem is often solved using the SP-problem (without negative weights) as an auxiliary procedure.

We conclude this section with one more application of matching theory to a problem concerning shortest paths, which is taken from [Gro85]. Consider a network $N=(G,w)$ on a graph G, where w is a nonnegative weight function. Let us call a path P in G *odd* if P contains an odd number of edges, so that P has odd length in the graph theoretical sense; *even* paths contain an even number of vertices.

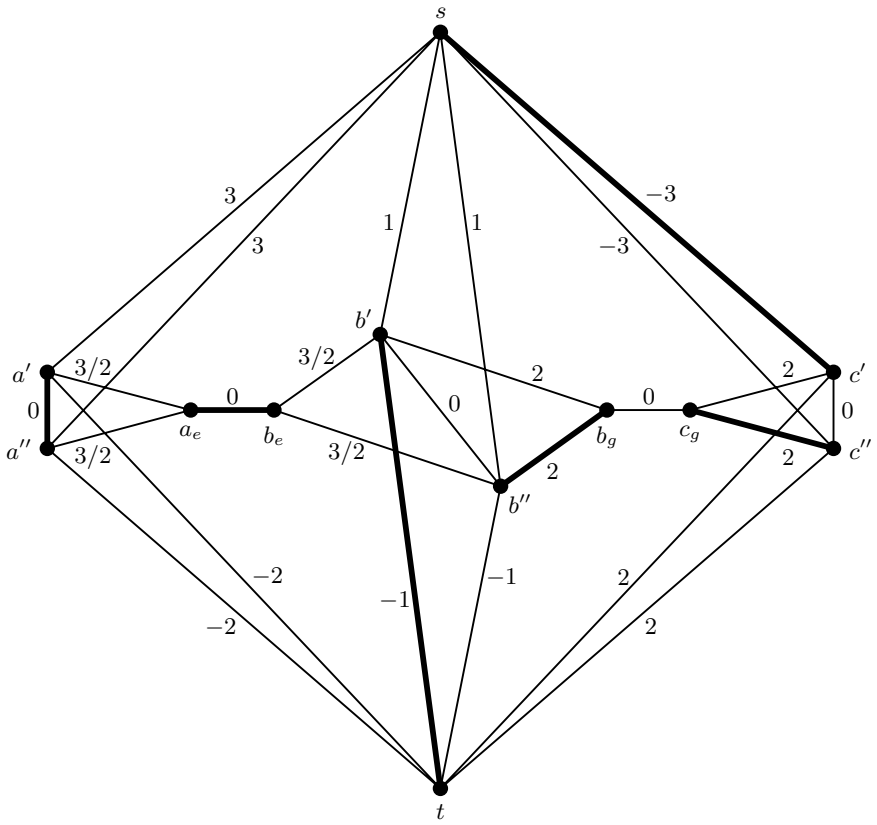


Fig. 14.12. A corresponding perfect matching in H

We want to find a shortest odd path between two given vertices s and t. This problem can be reduced to determining a perfect matching of minimal weight in a suitable auxiliary graph G', which again results from G by splitting vertices: each vertex $v \neq s, t$ of G is replaced by two vertices v' and v'', and an edge $v'v''$ of weight $w(v'v'') = 0$ is added to E. Moreover, each edge of G of the form sv or tv is replaced by the edge sv' or tv', respectively; and each edge uv with $u, v \neq s, t$ is replaced by two edges $u'v'$ and $u''v''$. Using similar arguments as for the proofs of Lemmas 14.6.1 and 14.6.2, one obtains the following result; the details will be left to the reader as an exercise.

Theorem 14.6.7. *Let $N = (G, w)$ be a network on a graph G, where w is a nonnegative weight function. Moreover, let s and t be two vertices of G, and let G' be the auxiliary graph described above. Then the odd $\{s, t\}$-paths P in G correspond bijectively to the perfect matchings M in G', and the length of P is equal to the weight of the matching M corresponding to P under this*

bijection. In particular, the shortest odd $\{s,t\}$-paths correspond bijectively to the perfect matchings of minimal weight in G'. □

Example 14.6.8. Let (G,w) be the network shown in Figure 14.13, where all edges $e \in E$ have weight $w(e) = 1$. Then the bold edges form an $\{s,t\}$-path

$$P: \; s \text{ — } u \text{ — } v \text{ — } t$$

of length 3, which corresponds to the perfect matching

$$K \; = \; \{su', u''v'', v't, a'a'', b'b'', c'c''\}$$

in the auxiliary graph G'; see Figure 14.14.

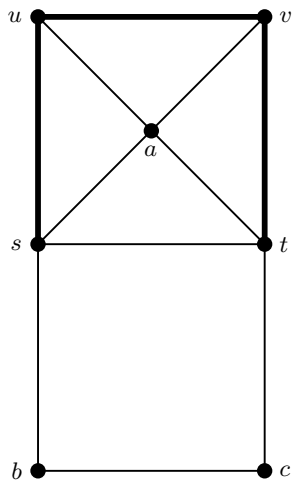


Fig. 14.13. A path of odd length in G

Exercise 14.6.9. Find a transformation similar to the one used in Theorem 14.6.7 which allows to find a shortest even $\{s,t\}$-path in (G,w) and apply this transformation to Example 14.6.8.

14.7 Some further problems

In this section, we briefly mention some further problems concerning matchings, beginning with problems with side constraints. Such problems occur in practice, for example, when planning the schedules for bus drivers, when designing school time tables, or even when analyzing bio-medical pictures; see [Bal85], [EvIS76], and [ItRo78]. We restrict our attention to rather simple – or at least seemingly simple – types of side constraints.

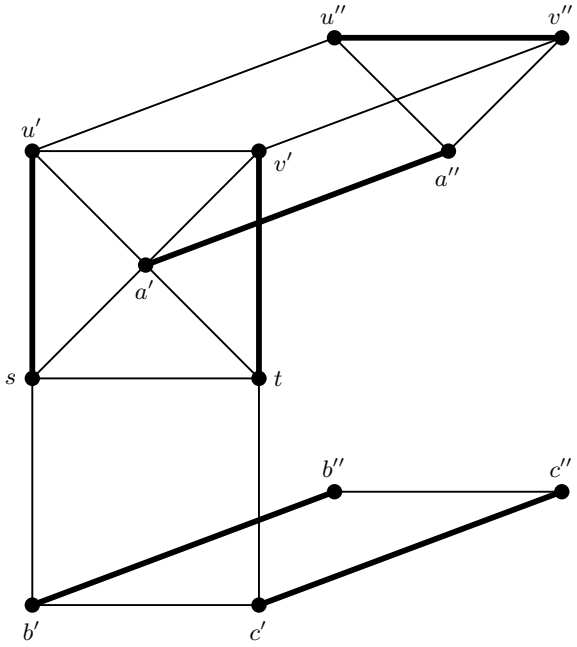


Fig. 14.14. The corresponding perfect matching in G'

Problem 14.7.1 (restricted perfect matching, RPM). Let $G = (V, E)$ be a graph, and let $E_1, \ldots, E_k$ be subsets of E and $b_1, \ldots, b_k$ be positive integers. Does there exist a perfect matching M of G satisfying the conditions

$$|M \cap E_i| \leq b_i \qquad \text{for } i = 1, \ldots, k? \tag{14.10}$$

If we want to fix the number k of constraints, we use the notation RPMk.

Exercise 14.7.2. Show that RPM1 can be solved with complexity $O(|V|^3)$ [ItRo78]. Hint: Reduce the problem to the determination of an optimal matching for the complete graph H on V with respect to a suitable weight function.

In contrast to the result of Exercise 14.7.2, the general problem RPM (that is, without restrictions on k) is NP-complete and thus probably not solvable in polynomial time; see [ItRT78]. The following related problem is rather interesting in this context.

Problem 14.7.3 (exact perfect matching, EPM). Let $G = (V, E)$ be a graph, and let R be a subset of E and b a positive integer. Does there exist a perfect matching M of G with $|M \cap R| = b$?

Exercise 14.7.4. Show that EPM is a special case of RPM2.

It is still unknown whether EPM (and RPM2, for that matter) admits a polynomial algorithm. However, it is known that the problem is polynomial at least for planar graphs; see Barahona and Pulleyblank [BaPu87]. Their algorithm is based on a result of Kasteleyn [Kas67] which allows one to determine the number of perfect matchings in a planar graph efficiently. It has been conjectured [PaYa82] that EPM is NP-complete for arbitrary graphs. For a good survey on exact matching problems, see [Lec86]; further information about EPM and various other problems with side constraints is contained in [Lec87].

Finally, we mention a rather different, but equally interesting, optimality criterion for perfect matchings: *stability*. In the bipartite case, the following interpretation is commonly used (the *stable marriage problem*): suppose there are n women and n men, who each rank the n persons of the opposite sex in a list. We want to find a perfect matching (which may be interpreted as a set of marriages) so that there is no unmarried couple consisting of a man and a woman who would both prefer each other to the partners they are married to according to the given matching.

Formally, we consider a weight function w on the complete orientation of $K_{n,n}$ for which the n edges having vertex v as tail are assigned a permutation of the numbers $1, \ldots, n$ as weights. We require a perfect matching M with the following property: if xy is an edge not contained in M and if xy' and $x'y$ are edges of M, then at least one of the two inequalities $w(xy') < w(xy)$ and $w(x'y) < w(xy)$ holds. Gale and Shapley [GaSh62] proved that, for each n and for each w, such a *stable matchings* exists and that a solution can be determined with complexity $O(n^2)$; see [Wil72], [Gus87], and [IrLG87].[8] Determining the number of all solutions, however, is an NP-hard problem; see [IrLe86].

The analogous problem for the complete graph K_{2n} (known as the *stable roommates problem*) is more difficult; for example, it cannot be solved for each choice of n and w. Irving [Irv85] gave an algorithm which decides with complexity $O(n^2)$ whether there exists a solution and, if this is the case, actually finds one; see also [Gus88]. We recommend the excellent monograph [GuIr89] for further study of this type of problems; see also [BaRa97] for a nice exposition.

[8] Stable matching problems – and some slight extensions, where one requires a prescribed number of edges for each vertex in one part of the graph – are not just of theoretical interest. Until recently, the Gale-Shapley algorithm was used to match medical students to hospital residencies, a rather difficult task if all parties are to be reasonably happy with the results (or, at the very least, to feel that they are subjected to a fair procedure); see the brief but interesting articles [Rob03a, Rob03b]. That an optimization algorithm is correct mathematically does, of course, not mean that it will achieve an end which is desirable from a social or moral point of view: there always is the problem of *what* should be optimized.

14.8 An application: Decoding graphical codes

In this final section, we consider an application of weighted matching, namely the decoding of graphical codes. We saw in Section 10.11 that graphical codes often are rather good binary codes, as far as their parameters are concerned. As mentioned there, one also needs an efficient encoding and decoding procedure if one actually wants to use such a code. We are now in a position to provide such a method. Again, we shall follow the tutorial paper [JuVa96].

We begin with the following efficient decoding algorithm for even graphical codes. We shall assume that $C = C_E(G)$ is a t-error correcting code, that is, $t \leq (g-1)/2$. Let X be the received word, and assume that at most t errors have occurred during transmission, so that $X = C+S$ for some even subgraph C of G and some subgraph S consisting of at most t edges. Note that S is an acyclic subgraph of G, since the girth of G is at least $2t+1$. Clearly, the odd degree pattern W of S (and hence of X) has weight $w \leq 2t$.

Thus we are faced with the following problem: given a set W consisting of an even number of vertices of G, find a spanning forest T – that is, an acyclic spanning subgraph – of G of smallest cardinality such that exactly the vertices of W have odd degree in T. But this is essentially a Chinese postman problem for the graph X,[9] and may thus be solved using the methods of Section 14.5. This idea was first suggested in [NtHa81], without giving any details. We present the following explicit algorithm taken from [JuVa97].

Algorithm 14.8.1. Consider an even graphical code $C = C_E(G)$ with parameters $[m, m-n+1, g]$ based on the connected graph G, where $g \geq 2t+1$. Let X be a word received and assume that at most t errors have occurred, that is, X is a subgraph of G of the form $X = C + S$ for some (unknown) even subgraph C of G and some (unknown) subgraph S consisting of at most t edges.

Procedure DECEVGC$(G, X; C)$

(1) Find the odd degree pattern W of X (by computing the degrees of all vertices in X); let $|W| = 2w$.
(2) For every pair $\{x, y\}$ of vertices in W, compute the distance $d(x, y)$ between x and y in G.
(3) Form the complete graph K on W.
(4) Find a minimum weight perfect matching $M = \{x_i y_i : i = 1, \ldots, w\}$ of K with respect to the weight function d computed in (2).
(5) Determine a path P_i of length $d(x_i, y_i)$ between x_i and y_i in G (for $i = 1, \ldots, w$).
(6) Let S be the symmetric difference of the paths $P_1, \ldots, P_w$.

[9] Our problem differs a little bit from the standard CPP for X, since we may use edges in G (and not only edges in X) to find a smallest subgraph with odd vertex pattern W. On the other hand, it is somewhat simpler since no edge weights are given: all edges have weight 1.

(7) Output $C = X + S$.

Theorem 14.8.2. *Algorithm 14.8.1 correctly decodes the even graphical code $C_E(G)$ in $O(tm + t^3)$ steps.*

Proof. The correctness of the decoding algorithm follows in a standard manner, as for the usual CPP. In order to analyze the complexity of this method, note $w \leq t$. Stage (1) clearly needs $O(m)$ steps. Applying breadth first search repeatedly, we may perform Stage 2 in $O(tm)$ steps; simultaneously, we may record information needed to actually compute the shortest paths between a given pair of vertices in W by storing suitable predecessor functions. Then Stages (5), (6) and (7) can be performed in $O(tm)$ steps. Finally, by Result 14.4.5, Stages (3) and (4) can be performed in $O(t^3)$ steps. □

If we consider t as fixed – which is quite usual in coding theory – Algorithm 14.8.1 is linear in the length of the code. In comparison, the original decoding algorithm suggested in [BrHa67] needed some precomputation and then still had a complexity quadratic in the length of the code.

The crucial idea in Algorithm 14.8.1 is the use of the odd degree pattern W of the subgraph X received for finding the error subgraph E, which just is the spanning forest of least weight which has the same vertices of odd degree as X. The general problem of finding a spanning forest of least weight with $2t$ prescribed vertices of odd degree is called the *t-join problem*. Note that in our case the t-join is uniquely determined from the vertices of odd degree (which does, of course, not hold in general). Hence it is conceivable that one might find an even more efficient algorithm to determine E from W for the special case we need, avoiding the costly CPP approach; this is still an open problem.

Before giving an example, let us also discuss the problem of decoding a general graphical code C^* based on a graph G. Clearly, this will depend on the odd pattern codes used in the constructions described in Theorems 10.11.13 and 10.11.15. For the sake of simplicity, we restrict ourselves to the method of Theorem 10.11.13; the other case is handled in a similar way [JuVa97].

Since we want to be able to correct up to t errors using C^*, we must first recognize if a received word R actually belongs to a codeword in $C_E(G)$ (possibly corrupted by up to t edges in error, resulting in an odd degree pattern W' of weight up to $2t$ for R) or to a word of the form $X = C + S$ with $C \in C_E(G)$ and $0 \neq W \in O$, where W denotes the odd degree pattern of X (again possibly corrupted by up to t edges in error, resulting in a word R which has an odd degree pattern W' of a weight reduced by up to $2t$ in comparison to that of W). Note that O is a $2t$-error correcting code, since it has minimum distance $2g \geq 4t + 2$ by the hypothesis of Theorem 10.11.13. Hence we can in fact (somehow) decode the received odd degree pattern R' into the correct odd degree pattern $W = R' + E'$. We may then make R into an even subgraph (using W and E) and decode this to determine C and finally X. More formally, we obtain the following procedure from [JuVa97].

Algorithm 14.8.3. Let C^* be a graphical $[m, m-n+1+k, g]$ code which is obtained from an even graphical $[m, m-n+1, g]$ code $C = C_E(G)$ (based on the connected graph G) by using an even binary $[n, k, 2g]$ code O as odd pattern code, where $g \geq 2t+1$. Let R be a word received and assume that at most t errors have occurred, so that R is subgraph of G of the form $R = C + S + E$ for some (arbitrary) even subgraph C of G, some (unknown) subgraph S with odd degree pattern W in O, and some (unknown) subgraph E consisting of at most t edges.

Procedure DECAUGGC$(G, R; X)$

(1) Find the received odd degree pattern R' of R (by computing the degrees of all vertices in R).
(2) Using the code O, decode R' into the correct odd degree pattern $W = R' + E' \in O$.
(3) Let S be some subgraph of G corresponding to the odd degree pattern W.
(4) Put $X' = R + S$ and decode X' into a subgraph $C \in C_E(G)$ using Algorithm 14.8.1.
(5) Output $X = C + S$.

Theorem 14.8.4. *Algorithm 14.8.3 correctly decodes the graphical code C^* in $O(tm + t^3) + O(D)$ steps, where $O(D)$ denotes the complexity of decoding O.*

Proof. We first show that the proposed decoding procedure is correct. By hypothesis, the error subgraph E consists of at most t edges. Hence the received odd degree pattern R' has the form $R' = W + E'$, where W is the correct odd degree pattern of the word $X = C + S$ transmitted and where E' is the odd degree pattern of E. Since E' has weight $2w$ for some integer $w \leq t$ and since O is a $2t$-error correcting code, we may indeed decode R' uniquely.

We now compute some subgraph S of G with the correct odd degree pattern W, e.g. by using the BFS-based procedure with complexity $O(m)$ outlined in Section 10.11; note that it is immaterial which specific choice of S is made. Since S belongs to C, the subgraph $X' = R + S$ still has the same error subgraph E, but its odd degree pattern is now simply E'. This shows that X' actually is obtained from a codeword in C corrupted by E, whence we have $X' = C + E$ for some $C \in C_E(G)$. Since E consists of at most t edges, Algorithm 14.8.1 can now be used to decode X' into C. The identity $R+S = C+E$ shows that Stage (5) indeed gives the correct codeword $X \in C^*$, proving the correctness of Algorithm 14.8.3.

Finally, the assertion on the complexity is clear in view of Theorem 14.8.2 and our remark concerning the realization of Stage (3). □

The quality of Algorithm 14.8.3 depends on the quality of the decoding algorithm available for decoding the odd pattern code O used. For instance, if we use an even graphical code for O, we can decode O itself by another application of Algorithm 14.8.1. But Algorithm 14.8.3 can also be quite effective

when other codes besides even graphical ones are used as odd pattern codes. In particular, one might choose O as a BCH-code of length n.[10]

We conclude this section with an example for decoding the augmented Petersen code P^* introduced in Example 10.11.11.

Example 14.8.5. We use the edge labelling of the Petersen graph given in Figure 14.15 for numbering the coordinate positions in the code P^* of Example 10.11.11.[11]

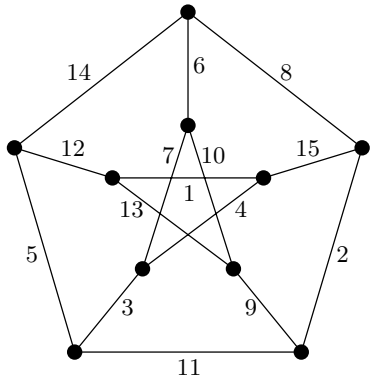


Fig. 14.15. An edge labelling of the Petersen graph

Let us assume that we have received the word $\mathbf{w} = (11111\,11100\,01001)$, i.e. the subgraph R in Figure 14.16 consisting of the bold edges; the four vertices of R having odd degree are labelled and drawn fat. As we have more even than odd vertices, the received word must belong to an even subgraph of P. In our case, the minimum weight matching obviously is given by the edges $4 = uv$ and $2 = xw$, since both of these two pairs of points are at distance 1. Hence we decode $\mathbf{w}$ to the codeword $(10101\,11100\,01001)$, i.e. the 8-cycle $\{1, 15, 8, 6, 7, 3, 5, 12\}$.

Exercise 14.8.6. Prove that the augmented Petersen code P^* is actually a *cyclic* code (provided that the coordinate positions are indexed by the edges

[10] The BCH codes constitute one of the most famous classes of codes; they are also used in many applications. We refer the reader to one of the standard text books in coding theory, for instance, [MacSl77], [vLi99] or [Bla83]. As is well-known, BCH-codes are efficiently decodable, either by the Berlekamp-Massey algorithm or the extended euclidean algorithm. Considering t as fixed, a complexity of $O(n \log n)$ can be achieved, which is comparable to our results when using an even graphical code as odd pattern code.

[11] This edge labelling was used in [JuVa96] to show that P^* is in fact isomorphic to a well-known example from coding theory, namely the double-error correcting BCH-code of length 15.

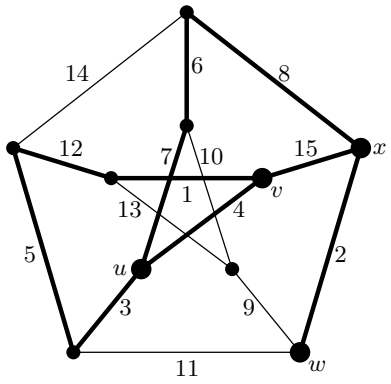


Fig. 14.16. Decoding the augmented Petersen code

in a suitable manner): whenever $\mathbf{x} = (x_1, \ldots, x_{15})$ is a codeword, the cyclic shift $(x_{15}, x_1, \ldots, x_{14})$ is also a codeword.

Hint: Use the edge labelling of the Petersen graph given in Figure 14.15 for indexing the coordinates, and note that it suffices to check that the cyclic shifts of the codewords in some basis of P^* are again in P^*.

15

A Hard Problem: The TSP

Which way are you goin'...
Jim Croce

Up to now, we have investigated only those optimization problems which allow an *efficient* – that is, polynomial – algorithm. In contrast, this final chapter will deal with a typical NP-complete problem: the travelling salesman problem already introduced in Chapter 1. We saw in Chapter 2 that no efficient algorithms are known for NP-complete problems, and that it is actually quite likely that no such algorithms can exist. Now we address the question of how such *hard* problems – which regularly occur in practical applications – might be approached: one uses, for instance, approximation techniques, heuristics, relaxations, post-optimization, local optima, and complete enumeration. We shall explain these methods only for the TSP, but they are typical for dealing with hard problems in general.

We will also briefly mention a further extremely important approach to solving hard problems: *polyhedral combinatorics*. A detailed discussion of this vast area of research would far exceed the limits of this book; as mentioned before, the reader can find an encyclopedic treatment of the polyhedral approach to combinatorial optimization in [Schr03].

The travelling salesman problem is one of the most famous and important problems in all of combinatorial optimization. It has been thoroughly studied for more than 60 years, and there is an abundance of literature on the subject, including quite a few books. Once again, our discussion can only scratch the surface of a vast area. For a more detailed study, we recommend the following three books: [LaLRS85], [GuPa02], and [ApBCC06]; the last of these opens with a very interesting chapter describing the origins and the history of the TSP, followed by an equally interesting chapter surveying the manyfold applications of the TSP in such diverse areas as logistics, genetics, telecommunications, and neuroscience.

15.1 Basic definitions

Let us start by recalling the formal definition of the TSP given in Section 1.4:

Problem 15.1.1 (travelling salesman problem, TSP). Let $w\colon E \to \mathbb{R}^+$ be a weight function on the complete graph K_n. We seek a cyclic permutation $(1, \pi(1), \ldots, \pi^{n-1}(1))$ of the vertex set $\{1, \ldots, n\}$ such that

$$w(\pi) \;=\; \sum_{i=1}^{n} w(\{i, \pi(i)\})$$

is minimal. We call any cyclic permutation π of $\{1, \ldots, n\}$ as well as the corresponding Hamiltonian cycle

$$1 \text{ — } \pi(1) \text{ — } \ldots \text{ — } \pi^{n-1}(1) \text{ — } 1$$

in K_n a *tour*; if $w(\pi)$ is minimal among all tours, π is called an *optimal tour*. The weights of the edges will be given via a matrix W, as explained in Section 1.4.

We shall use the following example also already introduced in Section 1.4 to illustrate various methods for finding a good solution of the TSP, which are the subject matter of this chapter.

Example 15.1.2. Determine an optimal tour for

$$\begin{array}{c} \\ Aa \\ Ba \\ Be \\ Du \\ Fr \\ Ha \\ Mu \\ Nu \\ St \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} Aa & Ba & Be & Du & Fr & Ha & Mu & Nu & St \end{array} \\
\left(\begin{array}{ccccccccc}
0 & 57 & 64 & 8 & 26 & 49 & 64 & 47 & 46 \\
57 & 0 & 88 & 54 & 34 & 83 & 37 & 43 & 27 \\
64 & 88 & 0 & 57 & 56 & 29 & 60 & 44 & 63 \\
8 & 54 & 57 & 0 & 23 & 43 & 63 & 44 & 41 \\
26 & 34 & 56 & 23 & 0 & 50 & 40 & 22 & 20 \\
49 & 83 & 29 & 43 & 50 & 0 & 80 & 63 & 70 \\
64 & 37 & 60 & 63 & 40 & 80 & 0 & 17 & 22 \\
47 & 43 & 44 & 44 & 22 & 63 & 17 & 0 & 19 \\
46 & 27 & 63 & 41 & 20 & 70 & 22 & 19 & 0
\end{array}\right)
\end{array}$$

We saw in Theorem 2.7.5 that the TSP is NP-complete, so that we cannot expect to find an efficient algorithm for solving it. Nevertheless, this problem is extremely important in practice, and techniques for solving – or at least approximately solving – instances of considerable size are essential.

Indeed, there are many applications of the TSP which bear little resemblance to the original *travelling salesman* interpretation. To mention a simple example, we might have to prepare the machines in a plant for n successive production processes. Let w_{ij} denote the setup cost arising if process j is scheduled immediately after process i; then the problem of finding an ordering for the n processes which minimizes the total setup cost can be viewed as a TSP. In [GrJR91] the reader can find an interesting practical case study, which demonstrates the relevance of approximation techniques for solving the TSP

to some tasks arising in the production of computers. A further impressive example is described in [BlSh89]: applying the TSP in X-ray crystallography resulted in dramatic savings in the amount of time a measuring process takes. Many further applications are discussed in [LeRi75], in [LaLRS85, Chapter 2], and in [ApBCC06, Chapter 2].

Note that the instance given in Example 15.1.2 has a rather special structure: the weights satisfy the triangle inequality $w_{ik} \leq w_{ij} + w_{jk}$. Of course, this holds whenever the weights stand for distances in the plane, or in a graph, and (more generally) whenever W corresponds to a metric space; see Section 3.2. Hence the following definition.

Problem 15.1.3 (metric travelling salesman problem, ΔTSP). Let $W = (w_{ij})$ be a symmetric matrix describing a TSP, and assume that W satisfies the triangle inequality:

$$w_{ik} \leq w_{ij} + w_{jk} \qquad \text{for } i, j, k = 1, \ldots, n.$$

Then one calls the given TSP *metric* or, for short, a ΔTSP.

Note that the TSP used in the proof of Theorem 2.7.5 is clearly metric. Hence we have the following result:

Theorem 15.1.4. *ΔTSP is NP-complete.* □

Nevertheless, the metric property does make a difference in the degree of complexity of a TSP: in the metric case, there always exists a reasonably good approximation algorithm; most likely, this does not hold for the general case, where the triangle inequality is not assumed; see Section 15.4.

Let us conclude this section with a brief discussion of three further variants of the TSP.

Problem 15.1.5 (asymmetric travelling salesman problem, ATSP). Instead of K_n, we consider the complete directed graph on n vertices: we allow the weight matrix W to be non-symmetric (but still with entries 0 on the main diagonal). This *asymmetric* TSP contains the usual TSP as a special case, and hence it is likewise NP-hard.

Example 15.1.6. We drop the condition that the travelling salesman should visit each city exactly once, so that we now consider not only Hamiltonian cycles, but also closed walks containing each vertex of K_n at least once. If the given TSP is metric, any optimal tour will still be an optimal solution. However, this does not hold in general, as the example given in Figure 15.1 shows: here (w, x, y, z, x, w) is a shortest closed walk (of length 6), but the shortest tour (w, x, y, z, w) has length 8.

Given a matrix $W = (w_{ij})$ not satisfying the triangle inequality, we may consider it as a matrix of lengths on K_n and then calculate the corresponding

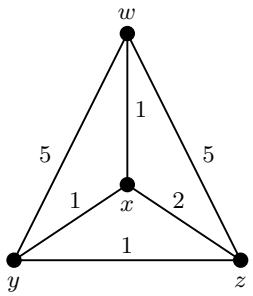


Fig. 15.1. A TSP for $n = 4$

distance matrix $D = (d_{ij})$. For example, we can use the algorithm of Floyd and Warshall for this purpose; see Section 3.9. Of course, D satisfies the triangle inequality and, hence, defines a metric TSP. It is easy to see that the optimal closed walks with respect to W correspond to the optimal tours with respect to D. Thus the seemingly more general problem described in Example 15.1.6 actually reduces to the metric TSP.

Finally, one may also consider an arbitrary connected graph G with some length function w instead of K_n. Then it is not at all clear whether any tours exist: we need to check first whether G is Hamiltonian. As proved in Section 2.8, this feasibility question is already an NP-complete problem in itself.

15.2 Lower bounds: Relaxations

From a practical point of view, it will often be necessary (and also sufficient) to construct a reasonably good approximate solution instead of an optimal tour. For example, it will suffice for most practical applications if we can provide an efficient method for finding a solution which is at most 2% worse than the optimal tour: using a vast amount of resources for further improvement of the quality of the solution would not make any economic sense. In this context, note also that input data – distances, for example – always have a limited accuracy, so that it might not even mean much to have a truly optimal solution at our disposal.

In order to judge the quality of an approximate solution, we need lower bounds on the length of a tour, and these bounds should not only be strong but also easily computable – aims which are, of course, usually contradictory. A standard approach is the use of suitable *relaxations*: instead of the original problem $\mathbf{P}$, we consider a problem $\mathbf{P}'$ containing $\mathbf{P}$; this auxiliary (simpler) problem is obtained by a suitable weakening of the conditions defining $\mathbf{P}$.

Then the weight $w(\mathbf{P}')$ of an optimal solution for $\mathbf{P}'$ is a lower bound for the weight $w(\mathbf{P})$ of an optimal solution for $\mathbf{P}$.[1]

Unfortunately, in many cases it is not possible to predict the quality of the approximation theoretically, so that we have to use empirical methods: for instance, comparing lower bounds found by relaxation with upper bounds given by solutions constructed by some heuristic. We shall consider various heuristics in Section 15.5; now we discuss several relaxations which have proved useful for dealing with TSP's. In this section, $\mathbf{P}$ is always a TSP on the complete graph K_n on $V = \{1, \ldots, n\}$, given by a weight matrix $W = (w_{ij})$.

A. The assignment relaxation

One choice for $\mathbf{P}'$ is the assignment problem AP defined in Example 7.4.12 and studied in Chapter 14. Thus we seek a permutation π of $\{1, \ldots, n\}$ for which $w_{1,\pi(1)} + \ldots + w_{n,\pi(n)}$ becomes minimal. In particular, we have to examine all cyclic permutations π (each of which determines a tour); for these permutations, the sum in question equals the length of the associated tour. Therefore we can indeed relax TSP to AP.

Note that we ought to be a little more careful here, since we should not just use the given matrix W to specify our AP: the diagonal entries $w_{ii} = 0$ would yield the identity as an optimal solution, which would result in a completely trivial lower bound: 0. As we are not interested in permutations with fixed points for the TSP anyway, we can avoid this problem by simply putting $w_{ii} = \infty$ for all i.[2] Clearly, this modification guarantees that an optimal solution of AP is a permutation without fixed points. If we should obtain a cyclic permutation as the optimal solution of AP, this permutation actually yields a solution of the TSP (by coincidence). Of course, in general, there is no reason why this should happen.

It is also comparatively easy to determine the weight $w(\mathrm{AP})$ of an optimal solution for the relaxed problem: the Hungarian algorithm of Section 14.2 will allow us to do so with complexity $O(n^3)$. Note that the Hungarian algorithm actually determines maximal weighted matchings, whereas we want to find a perfect matching of minimal weight for $K_{n,n}$ (with respect to the weights given by our modification of W). However, this merely requires a simple transformation, which was already discussed in Section 14.1.

It turns out that $w(\mathrm{AP})$ is usually a reasonably good approximation to $w(\mathrm{TSP})$ in practice – even though nobody has been able to prove this. Balas and Toth considered random instances for values of n between 40 and 100 and got an average of 82% of $w(\mathrm{TSP})$ for $w(\mathrm{AP})$; see [LaLRS85, Chapter 10]. That the assignment relaxation has such good approximation properties is,

[1] Our discussion refers to the TSP, but applies to minimization problems in general. Of course, with appropriate adjustments, it can also be transferred to maximization problems.

[2] In practice, this is done by using a sufficiently large number M instead of ∞: for instance, $M = \max\{w_{ij} : i, j = 1, \ldots, n\}$.

perhaps, to be expected, since the cyclic permutations form quite a big part of all permutations without fixed points: the number of permutations without fixed points in S_n is about $n!/e$, so that there is about one cyclic permutation among n/e fixed point free permutations; see, for example, [Hal86].

Balas and Toth examined the assignment relaxation also for the ATSP, using 400 problems randomly chosen in the range $50 \leq n \leq 250$. Here $w(\mathrm{AP})$ was on average $99,2\%$ of $w(\mathrm{ATSP})$.

Example 15.2.1. Consider the TSP of Example 15.1.2, where we replace the diagonal entries 0 in W by 88 (the maximum of the w_{ij}) to obtain the matrix W' for an associated AP. In order to reduce this AP to the determination of a maximal weighted matching, we consider the matrix $W'' = \left(88 - w'_{ij}\right)$ instead of W', as described in Section 14.1; note that W'' is the matrix given in Exercise 14.2.6. Then the Hungarian algorithm yields a maximal weighted matching, which has value 603; see the solution to 14.2.6. Any optimal matching for W'' is a solution of the original AP; hence $w(\mathrm{AP}) = 9 \times 88 - 603 = 189$. This gives the bound $w(\mathrm{TSP}) \geq 189$. As we will see, $w(\mathrm{TSP}) = 250$, so that the assignment relaxation amounts to less than 76% in this case.

Exercise 15.2.2. Try to provide an explanation for the phenomenon that the assignment relaxation tends to give much stronger bounds in the asymmetric case.

B. The MST relaxation

Now we use the problem MST of determining a minimal spanning tree of K_n (with respect to the weights given by W) as $\mathbf{P}'$. Of course, a tour is not a tree; but if we omit any edge from a tour, we indeed get a spanning tree, which is even of a very special type: it is a Hamiltonian path. This shows $w(\mathrm{MST}) \leq w(\mathrm{TSP})$. An optimal solution for MST can be determined with complexity $O(n^2)$, if we use the algorithm of Prim; see Theorem 4.4.4. We will prove later that this type of relaxation is rather good for the ΔTSP, whereas not much can be said for the general TSP; let us check how it works for our running example.

Example 15.2.3. For the TSP of Example 15.1.2, we obtain the minimal spanning tree T with $w(T) = 186$ shown in Figure 15.2. This bound is slightly inferior to the one provided in Example 15.2.1, but determining a minimal spanning tree is also much easier than solving an AP.

Note that the MST relaxation leaves out the weight of one of the edges of the tour, and hence is unnecessarily weak. This observation motivates a variation which we consider next.

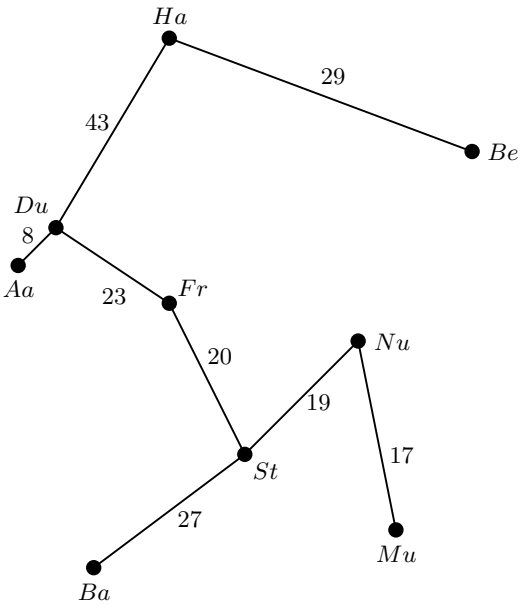


Fig. 15.2. MST relaxation

C. The s-tree relaxation

Let us choose a special vertex $s \in V$. An *s-tree* is a spanning tree for the induced subgraph $K_n \setminus s$ together with two edges incident with s.[3] Obviously, every tour is a special s-tree; hence $w(\mathrm{M}s\mathrm{T}) \leq w(\mathrm{TSP})$, where M$s$T denotes the problem of determining a minimal s-tree. Note that it is easy to solve this problem: just determine a minimal spanning tree for $K_n \setminus s$, and add those two edges incident with s which have smallest weight. Clearly, this can be done in $O(n^2)$ steps; see Theorem 4.4.4. Of course, the resulting bound will usually depend on the choice of the special vertex s.[4] As usual, let us apply this relaxation to our running example.

Example 15.2.4. We choose $s = Be$ in Example 15.1.2; this choice is motivated by the fact that the sum of the two smallest edge weights is maximal for this vertex. We obtain the s-tree B shown in Figure 15.3; note that B is the minimal spanning tree T given in Figure 15.2, with the edge $BeNu$ added. Hence $w(\mathrm{TSP}) \geq w(B) = 186 + 44 = 230$.

[3] In the literature, it is quite common to assume $s = 1$. Moreover, the term 1-*tree* is often used for the general concept (no matter which special vertex is selected), even though this is rather misleading.

[4] We might solve MsT for each choice of s to obtain the best possible bound, but this is probably not worth the extra effort provided that we select s judiciously.

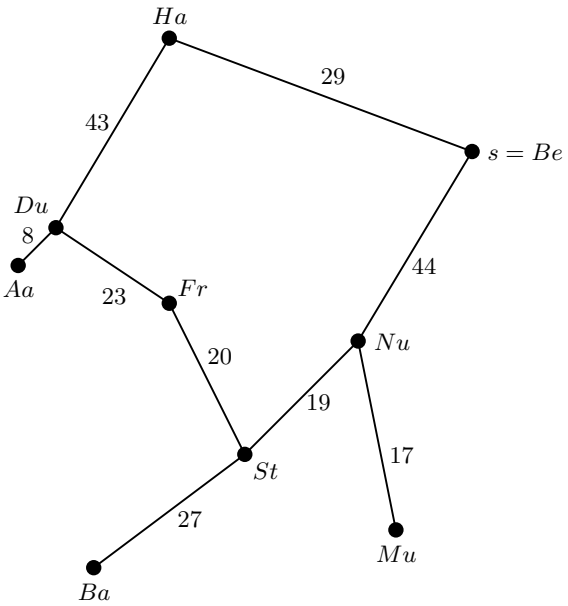


Fig. 15.3. s-tree relaxation

Exercise 15.2.5. Determine a minimal s-tree for the TSP of Example 15.1.2 for the other possible choices of the special vertex s.

Exercise 15.2.6. Discuss the relation between minimal spanning trees of K_n and minimal s-trees. In particular, find a condition on s which guarantees that a given minimal spanning tree of K_n extends to a minimal s-tree. Show that the strategy for selecting s which we have used in Example 15.2.4 does not always lead to a good bound.

Balas and Toth calculated the s-tree relaxation as well during their examination of the assignment relaxation. On average, $w(\mathrm{M}s\mathrm{T})$ was only 63% of $w(\mathrm{TSP})$, which is considerably worse than $w(\mathrm{AP})$. This may be explained by the fact that the number of s-trees is much larger than the number of permutations without fixed points.

Exercise 15.2.7. Determine the number of s-trees of K_n.
Hint: Use Corollary 1.2.11.

Exercise 15.2.8. For a vertex i, let $s(i)$ and $s'(i)$, respectively, denote the smallest and second smallest weight of an edge incident with i. Show $w(\mathrm{TSP}) \geq \frac{1}{2} \sum (s(i) + s'(i))$, and calculate the resulting bound for Example 15.1.2.

A variation of the s-tree relaxation may be found in [LeRe89]. In the next section, we will see that s-trees yield much better results when one also uses so-called *penalty functions*.

D. The LP relaxation

For the sake of completeness, we briefly discuss the relationship between the TSP and linear programming. By analogy with Example 14.3.1, the assignment relaxation of the TSP can be described by the following ZOLP:

$$\text{Minimize} \quad \sum_{i,j=1}^{n} w_{ij}x_{ij} \quad \text{subject to} \tag{15.1}$$

$$x_{ij} \in \{0,1\}, \quad \sum_{j=1}^{n} x_{ij} = 1 \quad \text{and} \quad \sum_{i=1}^{n} x_{ij} = 1 \quad (\text{for } i,j = 1,\ldots,n).$$

Then the admissible matrices (x_{ij}) correspond precisely to the permutations in S_n. In order to restrict the feasible solutions to tours, we add the following *subtour elimination constraints*:

$$\sum_{i,j\in S} x_{ij} \;\leq\; |S| - 1 \quad \text{for all} \;\; S \subset \{1,\ldots,n\}. \tag{15.2}$$

The inequalities (15.2) indeed have the effect that the path corresponding to the permutation has to leave the subset S, so that no cycles of a length smaller than n can occur.

Now let P be the polytope defined by the feasible solutions of (15.1) and (15.2); that is, the vertices of P correspond to the tours among the assignments. In principle, it is possible to describe P by a system of linear inequalities and solve the corresponding LP; unfortunately, the inequalities given in (15.1) and (15.2) do not suffice for this purpose.[5] Even worse, nobody knows a complete set of corresponding inequalities, although large classes of required inequalities (for example the *clique tree inequalities*) are known; see [LaLRS85, Chapter 8] and [GuPa02, Chapter 2], as well as [Nad90] and [BaFi93]. Also note that there is an exponential number of inequalities even in (15.2) alone.

Usually the inequalities in (15.1) are used together with some further (cleverly chosen) inequalities to define a first LP relaxation. In general, a whole sequence of such LP relaxations is solved, and the inequalities which are added for the next relaxation are chosen depending on the deficiencies of the solution calculated before, namely subtours or values $\neq 0, 1$. The most successful algorithms for solving large instances of the TSP use this approach; see, for

[5] The polytope P is the convex hull of the incidence vectors of tours: its vertices are 0-1-vectors. Leaving out the restriction $x_{ij} \in \{0,1\}$, the inequalities in (15.1) and (15.2) define a polytope P' containing P, which will (in general) have additional rational vertices. Thus all vertices of P' which are not 0-1-vectors have to be *cut off* by further appropriate inequalities.

example, [LaLRS85, Chapter 9], as well as [GrHo91], [PaRi87], [ApBCC95], [ApBCC03] and, in particular, the book [ApBCC06]. Finally, we also mention [PaSu91], where the quality of several formulations of the TSP as a linear program is studied.

15.3 Lower bounds: Subgradient optimization

In this section, we show how the lower bounds obtained from the s-tree relaxation can be improved considerably by using so-called *penalty functions*. This method was introduced by Held and Karp [HeKa70, HeKa71] and used successfully for solving comparatively large instances of the TSP. The basic idea is rather simple: we choose some vector $\mathbf{p} = (p_1, \ldots, p_n)^T \in \mathbb{R}^n$ and replace the weights w_{ij} of the given TSP by the transformed weights

$$w'_{ij} \;=\; w_{ij} + p_i + p_j \quad (i, j = 1, \ldots, n,\ i \neq j). \tag{15.3}$$

Let us denote the weight of a tour π with respect to the w'_{ij} by $w'(\pi)$. Clearly,

$$w'(\pi) \;=\; w(\pi) + 2(p_1 + \ldots + p_n) \quad \text{for every tour } \pi; \tag{15.4}$$

hence any tour which is optimal for W is optimal also for W'. On the other hand, the weight of an s-tree B is not transformed by just adding a constant:

$$w'(B) \;=\; w(B) + (p_1 \times \deg_B 1) + \ldots + (p_n \times \deg_B n)\,. \tag{15.5}$$

Thus the difference between the weight of a tour and the weight of an s-tree – which we would, of course, like to minimize – is

$$w'(\pi) - w'(B) \;=\; (w(\pi) - w(B)) - d_{\mathbf{p}}(B), \tag{15.6}$$

where

$$d_{\mathbf{p}}(B) \;=\; p_1\,(\deg_B 1 - 2) + \ldots + p_n\,(\deg_B n - 2)\,. \tag{15.7}$$

Let us assume that $d_{\mathbf{p}}(B)$ is positive for every s-tree B. Then we can improve the lower bound $w(\mathrm{M}s\mathrm{T})$ of the s-tree relaxation with respect to W by determining a minimal s-tree with respect to W': the gap between $w(\mathrm{TSP})$ and $w(\mathrm{M}s\mathrm{T})$ becomes smaller according to (15.6). We show below how this works for Example 15.1.2.

Of course, it is not clear whether such a vector $\mathbf{p}$ exists at all, and how it might be found. We will use the following simple strategy: calculate a minimal s-tree B_0 with respect to W, choose some positive constant c, and put

$$p_i \;=\; c \times \big(\deg_{B_0} i - 2\big) \quad \text{for } i = 1, \ldots, n. \tag{15.8}$$

Thus the non-zero coordinates of $\mathbf{p}$ impose a *penalty* on those vertices which do not have the correct degree 2 in B_0. This way of defining $\mathbf{p}$ has the following distinct advantage:

Exercise 15.3.1. Show that replacing W by W' according to the definition of $\mathbf{p}$ in (15.8) does *not* change the weight of a tour.

There remains the problem of choosing the value of c. It is possible to just use $c = 1$; however, in our example, we will select the most advantageous value (found by trial and error).

Example 15.3.2. Let B_0 be the minimal s-tree shown in Figure 15.3 for the TSP of Example 15.1.2, where $s = Be$. Note that the vertices Aa, Ba, and Mu have degree 1 in B_0, whereas the vertices Du, Nu, and St have degree 3 in B_0. Hence we obtain $\mathbf{p} = (-3, -3, 0, 3, 0, 0, -3, 3, 3)^T$, where we have chosen $c = 3$. This leads to the following transformed weight matrix W':

$$\begin{array}{c} \\ Aa \\ Ba \\ Be \\ Du \\ Fr \\ Ha \\ Mu \\ Nu \\ St \end{array} \begin{array}{c} \begin{array}{ccccccccc} Aa & Ba & Be & Du & Fr & Ha & Mu & Nu & St \end{array} \\ \left(\begin{array}{ccccccccc} 0 & 51 & 61 & 8 & 23 & 46 & 58 & 47 & 46 \\ 51 & 0 & 85 & 54 & 31 & 80 & 31 & 43 & 27 \\ 61 & 85 & 0 & 60 & 56 & 29 & 57 & 47 & 66 \\ 8 & 54 & 60 & 0 & 26 & 46 & 63 & 50 & 47 \\ 23 & 31 & 56 & 26 & 0 & 50 & 37 & 25 & 23 \\ 46 & 80 & 29 & 46 & 50 & 0 & 77 & 66 & 73 \\ 58 & 31 & 57 & 63 & 37 & 77 & 0 & 17 & 22 \\ 47 & 43 & 47 & 50 & 25 & 66 & 17 & 0 & 25 \\ 46 & 27 & 66 & 47 & 23 & 73 & 22 & 25 & 0 \end{array} \right) \end{array}$$

A minimal s-tree B_1 with respect to W' is displayed in Figure 15.4; its weight is $w'(B_1) = 242$. Note that we could also have used the edge $AaHa$ instead of $DuHa$. In this case, we would have obtained a different minimal s-tree, which would look less like a tour: also the vertices Aa and Du would have degree different from 2. For this reason, we prefer the tree B_1 of Figure 15.4. As the lengths of tours do not change for our choice of $\mathbf{p}$ (by Exercise 15.3.1), we have managed to improve the bound of Example 15.2.4 to $w(\mathrm{TSP}) \geq 242$.

As B_1 is not yet a tour, we try to continue in the same manner. Again, we select Be as the special vertex s. The vertices of B_1 which do not yet have the correct degree 2 are Ba and St. This time we choose $c = 4$ and $\mathbf{p} = (0, -4, 0, 0, 0, 0, 0, 0, 4)^T$, which yields the following weight matrix W'':

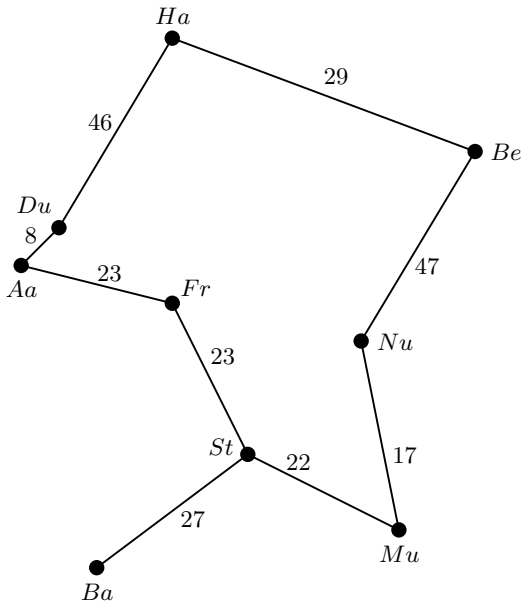


Fig. 15.4. Minimal s-tree with respect to W'

$$
\begin{array}{c}
 \\ Aa \\ Ba \\ Be \\ Du \\ Fr \\ Ha \\ Mu \\ Nu \\ St
\end{array}
\begin{array}{c}
\begin{array}{ccccccccc} Aa & Ba & Be & Du & Fr & Ha & Mu & Nu & St \end{array} \\
\left(\begin{array}{ccccccccc}
0 & 47 & 61 & 8 & 23 & 46 & 58 & 47 & 50 \\
47 & 0 & 81 & 50 & 27 & 76 & 27 & 39 & 27 \\
61 & 81 & 0 & 60 & 56 & 29 & 57 & 47 & 70 \\
8 & 50 & 60 & 0 & 26 & 46 & 63 & 50 & 51 \\
23 & 27 & 56 & 26 & 0 & 50 & 37 & 25 & 27 \\
46 & 76 & 29 & 46 & 50 & 0 & 77 & 66 & 77 \\
58 & 27 & 57 & 63 & 37 & 77 & 0 & 17 & 26 \\
47 & 39 & 47 & 50 & 25 & 66 & 17 & 0 & 29 \\
50 & 27 & 70 & 51 & 27 & 77 & 26 & 29 & 0
\end{array}\right)
\end{array}
$$

A minimal s-tree B_2 with respect to W'' is shown in Figure 15.5. (Looking at the degrees, we find it advisable to include the edge $BaSt$ instead of either $BaMu$ or $BaFr$.) This improves our bound to $w(\text{TSP}) \geq w''(B_2) = 248$.

Again, there are two vertices which do not yet have the correct degree 2, namely Ba and Mu. This time we choose $c = 1$ and $\mathbf{p}$=$(0, -1, 0, 0, 0, 0, 1, 0, 0)$. We leave it to the reader to compute the corresponding weight matrix W^* and to check that this leads to the minimal s-tree B_3 of weight $w^*(B_3) = 250$ shown in Figure 15.6.

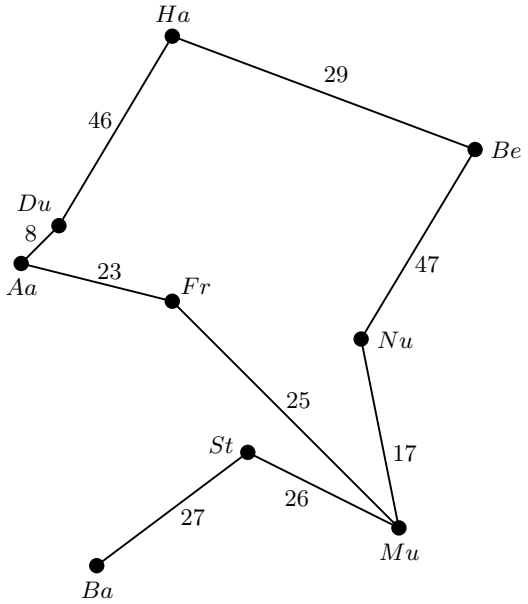


Fig. 15.5. Minimal s-tree with respect to W''

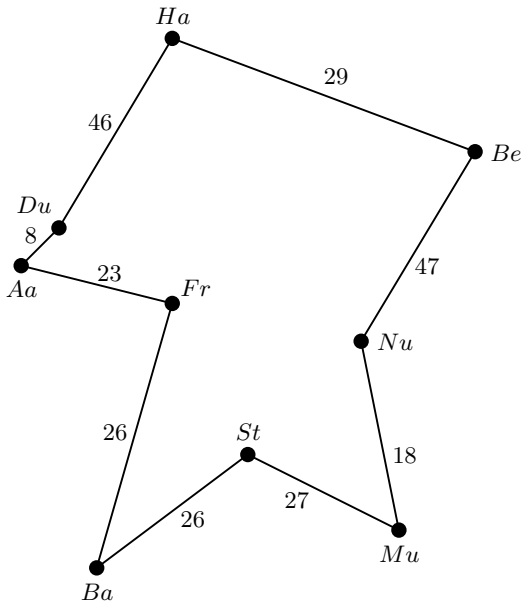


Fig. 15.6. Minimal s-tree with respect to W^*: an optimal tour

As B_3 is actually a tour, we have now (coincidentally) *solved* the TSP of Example 15.1.2: the tour

$$Aa \text{ — } Du \text{ — } Ha \text{ — } Be \text{ — } Nu \text{ — } Mu \text{ — } St \text{ — } Ba \text{ — } Fr \text{ — } Aa$$

is optimal, and hence $w(\mathrm{TSP}) = 250$.

Of course, it would be nice to be able to choose the vector $\mathbf{p}$ as advantageously as possible. As a tour which is optimal with respect to w is also optimal with respect to w', where w' is defined as in (15.3), we want to minimize the gap $d(\mathbf{p})$ between the length $w'(\mathrm{TSP})$ of an optimal tour and the weight $w'(B)$ of a minimal s-tree B. Equations (15.5) and (15.6) yield

$$d(\mathbf{p}) \;=\; w(\mathrm{TSP}) - \min\{w(B) + d_{\mathbf{p}}(B) \colon B \text{ is an } s\text{-tree}\}\,.$$

If we want to minimize $d(\mathbf{p})$, we need to determine

$$(L) \qquad L(w) \;=\; \max\big\{\min\{w(B) + d_{\mathbf{p}}(B) \colon B \text{ is an } s\text{-tree}\} \colon \mathbf{p} \in \mathbb{R}^n\big\}.$$

In general, we will not end up with $L(w) = w(\mathrm{TSP})$: it is quite possible that no choice of $\mathbf{p}$ yields a minimal s-tree which is already a tour; an example for this situation can be found in [HeKa70]. But the lower bound for $w(\mathrm{TSP})$ given by (L) is particularly strong: the values of $L(w)$ are on average more than 99% of $w(\mathrm{TSP})$ according to [VoJo82]. Interesting theoretical studies of the Held-Karp technique are due to [Wol80] and [ShWi90]; these authors proved that, in the metric case, the weight of an optimal tour is bounded by 3/2 times the Held-Karp lower bound. See also [ChGK06] and the references given there for more recent work on the Held-Karp bound, also for the metric ATSP.

Of course, solving (L) is a considerably more involved problem than the original s-tree relaxation. There are various approaches to solving problems of this type; the vectors $\mathbf{p}$ are called *subgradients* in the general context. These subgradients can be used for solving (L) recursively; this yields a method which is guaranteed to converge to $L(w)$ (for an appropriate choice of the step widths c). Unfortunately, one cannot predict how many steps will be required, so that the process is often terminated in practice as soon as the improvement between successive values becomes rather small. Fortunately, Held and Karp showed that (L) can also be formulated in terms of linear programming, and this yields, in practical applications, good bounds with moderate effort.

The problem (L) is a special case of a much more general method which is used quite often for integer linear programming problems: *Lagrange relaxation*; we refer to [Sha79] and [Fis81]. The approach via *subgradient optimization* is only one of several ways to solve Lagrange relaxations; it is described in detail (together with other methods) in [Sho85]; see also [HeWC74].

Appropriate relaxations are very important for finding the optimal solution of a TSP, because they form an essential part of *branch-and-bound* techniques; we will present an example for such a method in Section 15.8. We

refer the reader to [VoJo82] and to [LaLRS85, Chapter 10] for more detailed information. Further methods for determining lower bounds can be found, for example, in [CaFT89].

15.4 Approximation algorithms

The preceding two sections treated the problem of finding lower bounds on the length of an optimal tour, so it is now natural to ask for upper bounds. It would be nice to have an algorithm (of small complexity, if possible) for constructing a tour which always gives a provably good approximation to the optimal solution. We need a definition to make this idea more precise, which generalizes the approach we took when we studied the greedy algorithm as an approximation method in Section 5.4.

Let **P** be an optimization problem, and let **A** be an algorithm which calculates a feasible – though not necessarily optimal – solution for any given instance I of **P**. We denote the weights of an optimal solution and of the solution constructed by **A** by $w(I)$ and $w_{\mathbf{A}}(I)$, respectively. If the inequality

$$|w_{\mathbf{A}}(I) - w(I)| \;\leq\; \varepsilon w(I) \tag{15.9}$$

holds for each instance I, we call **A** an *ε-approximative algorithm* for **P**. For example, a 1-approximative algorithm for the TSP would always yields a tour which is at most twice as long as an optimal tour.

Given an NP-complete problem, there is little hope to find a polynomial algorithm which solves **P** correctly. Thus it seems promising to look instead for a polynomial ε-approximative algorithm, with ε as small as possible. Unfortunately, this approach is often just as difficult as solving the original problem. In particular, this holds for the TSP, as the following result of Sahni and Gonzales [SaGo76] shows.

Theorem 15.4.1. *If there exists an ε-approximative polynomial algorithm for the TSP, then $P = NP$.*

Proof. Let **A** be an ε-approximative polynomial algorithm for the TSP. We will use **A** to construct a polynomial algorithm for determining a Hamiltonian cycle; then the assertion follows from Theorem 2.7.4. The construction resembles the one given in the proof of Theorem 2.7.5. Let $G = (V, E)$ be a connected graph, and consider the complete graph K_V on V with weights

$$w_{ij} \;=\; \begin{cases} 1 & \text{for } ij \in E \\ 2 + \varepsilon|V| & \text{otherwise.} \end{cases}$$

If the given algorithm **A** should determine a tour of weight $n = |V|$ for this instance of the TSP, then G is obviously Hamiltonian.

Conversely, suppose that G contains a Hamiltonian cycle. Then the corresponding tour has weight n and is trivially optimal. As $\mathbf{A}$ is ε-approximative by hypothesis, it will compute a tour π of weight $w(\pi) \leq (1+\varepsilon)n$. Suppose that π contains an edge $e \notin E$. Then

$$w(\pi) \;\geq\; (n-1) + (2+\varepsilon n) \;=\; (1+\varepsilon)n + 1,$$

a contradiction. Hence the tour π determined by $\mathbf{A}$ actually induces a Hamiltonian cycle in G, so that it has in fact weight n.

We have proved that G is Hamiltonian if and only if $\mathbf{A}$ constructs a tour of weight n for our auxiliary TSP, so that $\mathbf{A}$ would indeed yield a polynomial algorithm for HC. □

Clearly, a result analogous to Theorem 15.4.1 holds for the ATSP. Interestingly, the situation is much more favorable for the metric TSP. We need a definition and a lemma. Let K_n be the complete graph on $V = \{1, \ldots, n\}$. Then any connected Eulerian multigraph on V is called a *spanning Eulerian multigraph* for K_n.

Lemma 15.4.2. *Let W be the weight matrix of a ΔTSP on K_n, and let $G = (V, E)$ be a spanning Eulerian multigraph for K_n. Then one can construct with complexity $O(|E|)$ a tour π satisfying $w(\pi) \leq w(E)$.*

Proof. By Example 2.5.3, it is possible to determine with complexity $O(|E|)$ an Euler tour C for G. Write the sequence of vertices corresponding to C in the form $(i_1, P_1, i_2, P_2, \ldots, i_n, P_n, i_1)$, where $(i_1, \ldots, i_n)$ is a permutation of $\{1, \ldots, n\}$ and where the $P_1, \ldots, P_n$ are (possibly empty) sequences on $\{1, \ldots, n\}$. Then $(i_1, \ldots, i_n, i_1)$ is a tour π satisfying

$$w(\pi) \;=\; \sum_{j=1}^{n} w_{i_j i_{j+1}} \;\leq\; w(E) \quad (\text{where } i_{n+1} = i_1),$$

since the sum of the weights of all edges in a path from x to y is always an upper bound for w_{xy}[6] and since each edge occurs exactly once in the Euler tour C. □

We now construct spanning Eulerian multigraphs of small weight and use these to design approximative algorithms for the metric TSP. The easiest method is simply to double the edges of a minimal spanning tree, which results in the following well-known algorithm.

Algorithm 15.4.3 (tree algorithm). Let $W = (w_{ij})$ be the weight matrix for a ΔTSP on K_n.

(1) Determine a minimal spanning tree T for K_n (with respect to the weights given by W).

[6] Note that this is the one point in the proof where we make use of the triangle inequality.

(2) Let $G = (V, E)$ be the multigraph which results from replacing each edge of T with two parallel edges.
(3) Determine an Euler tour C for G.
(4) Choose a tour contained in C (as described in the proof of Lemma 15.4.2).

Example 15.4.4. Let us again consider Example 15.1.2. We saw in Example 15.2.3 that the MST relaxation yields the minimal spanning tree T of weight $w(T) = 186$ displayed in Figure 15.2. A possible Euler tour for the doubled tree is

$$(Aa, Du, Ha, Be, Ha, Du, Fr, St, Ba, St, Nu, Mu, Nu, St, Fr, Du, Aa),$$

which contains the tour

$$\pi : \ Aa — Du — Ha — Be — Fr — St — Ba — Nu — Mu — Aa$$

of length 307; see Figure 15.7. Note that Theorem 15.4.5 below only guarantees that we will be able to find a tour of length ≤ 372; it is just good luck that π is actually a considerably better solution.

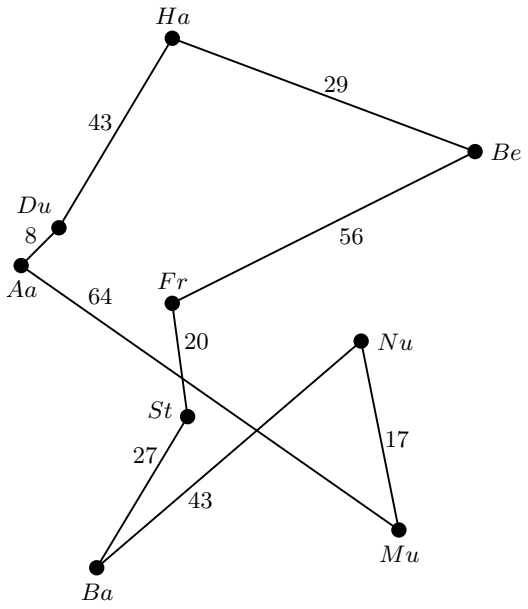


Fig. 15.7. Tour constructed by the tree algorithm

Theorem 15.4.5. *Algorithm 15.4.3 is a* 1*-approximative algorithm of complexity* $O(n^2)$ *for* ΔTSP.

Proof. Using the algorithm of Prim, step (1) has complexity $O(n^2)$; see Theorem 4.4.4. The procedure EULER developed in Chapter 2 can be used to perform step (3) in $O(|E|) = O(n)$ steps. Clearly, steps (2) and (4) also have complexity $O(n)$. This establishes the desired complexity bound.

By Lemma 15.4.2, the tree algorithm constructs a tour π with weight $w(\pi) \leq 2w(T)$. On the other hand, the MST relaxation of Section 15.2 shows that all tours have weight at least $w(T)$. Hence $w(\pi)$ is indeed at most twice the weight of an optimal tour. □

It is quite possible that Algorithm 15.4.3 constructs a tour whose weight is close to $2w(\mathrm{TSP})$; see [LaLRS85, Chapter 5]. In contrast, the difference between the length of the tour of Example 15.4.4 and the optimal tour of Example 15.3.2 is less than 23%.

Next we present a $\frac{1}{2}$-approximative algorithm, which is due to Christofides [Chr76]; his method is a little more involved.

Algorithm 15.4.6 (Christofides' algorithm). Let $W = (w_{ij})$ be a weight matrix for a ΔTSP on K_n.

(1) Determine a minimal spanning tree T of K_n (with respect to W).
(2) Let X be the set of all vertices which have odd degree in T.
(3) Let H be the complete graph on X (with respect to the weights given by the relevant entries of W).
(4) Determine a perfect matching M of minimal weight in H.
(5) Let $G = (V, E)$ be the multigraph which results from adding the edges of M to T.
(6) Determine an Euler tour C of G.
(7) Choose a tour contained in C (as described in the proof of Lemma 15.4.2).

Theorem 15.4.7. *Algorithm 15.4.6 is a $\frac{1}{2}$-approximative algorithm of complexity $O(n^3)$ for ΔTSP.*

Proof. In addition to the procedures also used in Algorithm 15.4.3, Algorithm 15.4.6 requires the determination of a perfect matching of minimal weight. This can certainly be done with complexity $O(n^3)$ (by Result 14.4.5), so that the total complexity will be $O(n^3)$. It remains to consider the quality of the resulting approximation.

As G is Eulerian by Theorem 1.3.1, the tour π determined in step (5) satisfies the inequality

$$w(\pi) \leq w(E) = w(T) + w(M) \tag{15.10}$$

(by Lemma 15.4.2). Thus we have to find a bound for $w(M)$. Write $|X| = 2m$ and let $(i_1, i_2, \ldots, i_{2m})$ be the vertices of X in the order in which they occur in some optimal tour σ.[7] We consider the following two matchings of H:

[7] Of course we do not know such an optimal tour explicitly, but that does not matter for our argument.

$$M_1 = \{i_1i_2, i_3i_4, \ldots, i_{2m-1}i_{2m}\} \quad \text{and} \quad M_2 = \{i_2i_3, i_4i_5, \ldots, i_{2m}i_1\}.$$

The triangle inequality for W implies

$$\begin{aligned} w(\sigma) &\geq w_{i_1i_2} + w_{i_2i_3} + \ldots + w_{i_{2m-1}i_{2m}} + w_{i_{2m}i_1} \\ &= w(M_1) + w(M_2) \geq 2w(M), \end{aligned}$$

since M is a perfect matching of minimal weight. Hence

$$w(M) \leq w(\text{TSP})/2 \qquad \text{and} \qquad w(T) \leq w(\text{TSP})$$

(by the MST relaxation), and (15.10) yields $w(\pi) \leq 3w(\text{TSP})/2$. □

The bound of Theorem 15.4.7 is likewise best possible: there are examples where Christofides' algorithm constructs a tour π for which the ratio between $w(\pi)$ and $w(\text{TSP})$ is arbitrarily close to 3/2; see [CoNe78].

Example 15.4.8. Consider once again Example 15.1.2, and let T be the minimal spanning tree with weight $w(T) = 186$ given in Example 15.2.3. The set of vertices of odd degree is $X = \{Aa, Be, Du, St, Ba, Mu\}$. Thus we require a perfect matching M of minimal weight with respect to the following matrix:

$$\begin{array}{c} \\ Aa \\ Ba \\ Be \\ Du \\ Mu \\ St \end{array} \begin{array}{c} \begin{array}{cccccc} Aa & Ba & Be & Du & Mu & St \end{array} \\ \begin{pmatrix} - & 57 & 64 & 8 & 64 & 46 \\ 57 & - & 88 & 54 & 37 & 27 \\ 64 & 88 & - & 57 & 60 & 63 \\ 8 & 54 & 57 & - & 63 & 41 \\ 64 & 37 & 60 & 63 & - & 22 \\ 46 & 27 & 63 & 41 & 22 & - \end{pmatrix} \end{array}$$

By inspection, we obtain $M = \{AaDu, BeMu, BaSt\}$ with $w(M) = 95$. Adding the edges of M to T yields an Eulerian multigraph of weight 281 with Euler tour

$$(Be, Mu, Nu, St, Ba, St, Fr, Du, Aa, Du, Ha, Be),$$

which contains the tour

$$Be — Mu — Nu — St — Ba — Fr — Du — Aa — Ha — Be;$$

see Figure 15.8. Note that this tour has weight 266, which is only 6% more than the optimal value of 250.

The algorithm of Christofides is the best approximative algorithm for the ΔTSP known so far: the existence of an ε-approximative polynomial algorithm for the ΔTSP for some $\varepsilon < 1/2$ is open. A priori, it would be conceivable that

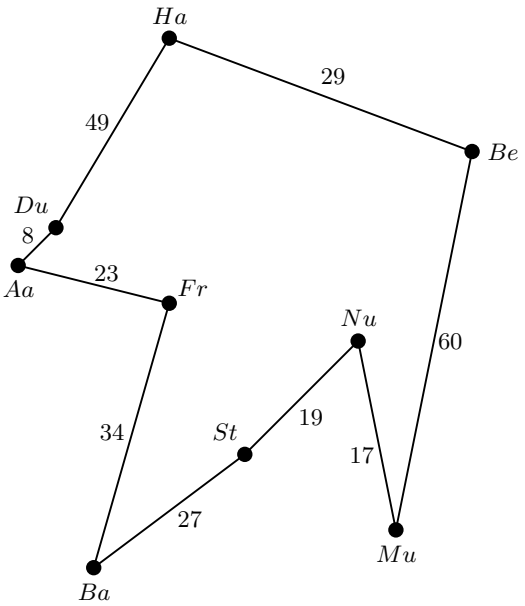


Fig. 15.8. Tour constructed by Christofides' algorithm

such an algorithm exists for each $\varepsilon > 0$; such a family of algorithms is called a (polynomial time) *approximation scheme*.

It was known for many years that P $\neq$ NP would imply the nonexistence of a *fully polynomial* approximation scheme for the ΔTSP: there is no family of ε-approximative algorithms such that their complexity is polynomial in n and $1/\varepsilon$; see Theorem 6 in [LaLRS85, Chapter 5]. Later this result was extended to arbitrary approximation schemes; see [ArLMS92]; this holds even if the weights are restricted 1 and 2. More recently, Papadimitriou and Venkala [PaVe06] proved that the metric TSP cannot admit an ε-approximative polynomial algorithm for some $\varepsilon < 1/219$, unless P = NP; they also obtained a similar result – with $\varepsilon < 1/116$ – for the metric case of the ATSP. It is a major open problem whether or not the metric ATSP admits a polynomial ε-approximative algorithm for any $\varepsilon > 0$.[8]

Nevertheless, there are also some positive results. Papadimitriou and Yannakakis [PaYa93] managed to find an $\frac{1}{6}$-approximative algorithm for the interesting special case of the ΔTSP with weights restricted to 1 and 2, which

[8] Some other important problems are even more difficult to handle than the ΔTSP. For example, the existence of a polynomial ε-approximative algorithm for determining a maximal clique (for *any* particular choice of $\varepsilon > 0$) already implies P = NP; see [ArSa02]. For even stronger results in this direction, we refer to [Zuc96]. All these results use an interesting concept from theoretical computer science: so-called *transparent proofs*; see, for example, [BaFL91] and [BaFLS91].

occurred in the proof of Theorem 2.7.5. Even more striking is the fact that the *Euclidean TSP* – in particular, the special case of the symmetric TSP in the plane, with the standard Euclidean distance – admits an approximation scheme; see [Aro98] and [Mit99].

To close this section, we use the main idea behind Algorithm 15.4.3 to prove the simple bound mentioned in Section 4.6 for the ratio of the weight of a minimal Steiner tree to a minimal spanning tree; this result is due to E. F. Moore (see [GiPo68]).

Theorem 15.4.9. *Let $v_1, \ldots, v_n$ be n points in the Euclidean plane, and let S and T be a minimal Steiner tree and a minimal spanning tree, respectively, for these n points. Then $w(T) \leq 2w(S)$, where the weight $w(uv)$ of an edge uv is the Euclidean distance between u and v.*

Proof. Consider the ΔTSP on $V = \{v_1, \ldots, v_n\}$ with the Euclidean distance as weight function. As in Algorithm 15.4.3, we double the edges of a minimal Steiner tree S for $v_1, \ldots, v_n$ and determine an Euler tour C for the resulting Eulerian multigraph (V, E). As S contains the vertices $v_1, \ldots, v_n$ (perhaps together with some Steiner points), it is possible to choose a tour π contained in C. As in Lemma 15.4.2, one shows

$$w(\pi) \leq w(E) = 2w(S).$$

But then

$$w(T) \leq w(\mathrm{TSP}) \leq w(\pi) \leq 2w(S)$$

is immediate. □

Note that the preceding proof of Theorem 15.4.9 is also valid for the Steiner problem in an arbitrary metric space.

15.5 Upper bounds: Heuristics

We saw in Theorem 15.4.1 that we cannot expect to find good approximative algorithms for the general TSP. Still, we would like to be able to solve a given TSP as well as possible. After having found lower bounds for $w(\mathrm{TSP})$ in Sections 15.2 and 15.3, we now look more closely at the problem of determining upper bounds. Of course, *any* tour yields an upper bound. As a tour chosen randomly cannot be expected to give a very good bound, one usually resorts to heuristics for constructing suitable tours. Of course, these heuristics might also produce rather weak bounds, but we may at least hope for a meaningful result. It is also common practice to try to improve a candidate tour (whether constructed by heuristic methods or at random) by some sort of post-optimization procedure; we will consider this approach in the next section.

Perhaps the most frequently used heuristics are the so-called *insertion algorithms*. Such an algorithm first chooses an arbitrary city x_1 as a starting point for the tour to be constructed. Then a city x_2 is chosen – using some criterion still to be specified – and added to the partial tour constructed so far, giving the partial tour (x_1, x_2, x_1). This procedure is repeated until a tour $(x_1, \ldots, x_n, x_1)$ is obtained. Thus the current partial tour of length k is always extended to a tour of length $k+1$ by adding one more city in the k-th iteration; this involves two tasks:

(a) choosing the city to be added and
(b) deciding where the city chosen in (a) will be inserted into the current partial tour.

There are several standard strategies for choosing the city in (a): arbitrary choice; selecting the city which has maximal (or, alternatively, minimal) distance to the cities previously chosen; or choosing the city which is cheapest to add. We also have to settle on a criterion for step (b); here an obvious strategy is to insert the city at that point of the partial tour where the least additional cost occurs.

We describe an algorithm which usually works quite nicely in practice, although there are no bounds known for its quality – not even in the metric case. In step (a), we always choose the city which has maximal distance to the current partial tour. This might appear strange at first glance, but there is a good reason for this strategy: as all the cities have to appear in the tour anyway, it seems best to plan the rough outline of the tour first, by taking all those cities into account which are far apart from each other. Towards the end of the insertion process, the remaining cities merely change the details of the tour, which should not increase the cost that much any more.

Thus we choose in step (a) that city y which has maximal distance to the partial tour $\pi = (x_1, \ldots, x_k, x_1)$. Here the *distance* of y to π is defined as $d(y) = \min\{w_{y,x_1}, \ldots, w_{y,x_k}\}$. In step (b), the selected city y is then inserted between two consecutive cities i and j in π for which the cost

$$c(i,j) \;=\; w_{i,y} + w_{y,j} - w_{i,j}$$

caused by the insertion is minimal. We obtain the following algorithm, where distances are stored in an array denoted by d.

Algorithm 15.5.1 (farthest insertion). Let $W = (w_{ij})$ be the weight matrix of a TSP on K_n, and let s be the vertex of K_n chosen as the starting point of the tour C to be constructed.

Procedure FARIN$(W, s; C)$

(1) $C \leftarrow (s, s)$, $K \leftarrow \{ss\}$, $w \leftarrow 0$;
(2) **for** $u = 1$ **to** n **do** $d(u) \leftarrow w_{su}$ **od**
(3) **for** $k = 1$ **to** $n-1$ **do**
(4) choose y with $d(y) = \max\{d(u)\colon u = 1, \ldots, n\}$;

(5) **for** $e = ij \in K$ **do** $c(e) \leftarrow w_{i,y} + w_{y,j} - w_{i,j}$ **od**
(6) choose an edge $f \in K$ with $c(f) = \min\{c(e) : e \in K\}$, say $f = uv$;
(7) insert y between u and v in C;
(8) $K \leftarrow (K \setminus \{f\}) \cup \{uy, yv\}$, $w \leftarrow w + c(f)$, $d(y) \leftarrow 0$;
(9) **for** $x \in \{1, \ldots, n\} \setminus C$ **do** $d(x) \leftarrow \min\{d(x), w_{yx}\}$
(10) **od**

The simple proof of the following theorem will be left to the reader.

Theorem 15.5.2. *Algorithm 15.5.1 constructs with complexity $O(n^2)$ a tour C with weight $w(C) = w$.* □

Example 15.5.3. Consider again the TSP of Example 15.1.2; we choose the vertex $s = Fr$ as our starting point. We always state the distances in the form of a 9-tuple d containing the distances to $Aa, \ldots, St$ (in this order).

In the first iteration, we obtain $d = (26, 34, 56, 23, 0, 50, 40, 22, 20)$. The vertex of maximal distance is Be with $d(Be) = 56$. Thus the partial tour for $i = 1$ is $T = (Fr, Be, Fr)$ of length 112.

The distances in the second iteration are $(26, 34, 0, 23, 0, 29, 40, 22, 20)$; this yields $y = Mu$, $d(Mu) = 40$ and $C = (Fr, Be, Mu, Fr)$ with length $w = 156$.

For $k = 3$, the distances are $(26, 34, 0, 23, 0, 29, 0, 17, 20)$; hence $y = Ba$ and $C = (Fr, Be, Mu, Ba, Fr)$ with length $w = 187$.

In the fourth iteration, $d = (26, 0, 0, 23, 0, 29, 0, 17, 20)$. Inserting $y = Ha$ at the point of least cost yields $C = (Fr, Ha, Be, Mu, Ba, Fr)$ and $w = 210$.

For $k = 5$, we obtain $d = (26, 0, 0, 23, 0, 0, 0, 17, 20)$. Now $y = Aa$, and we obtain $C = (Fr, Aa, Ha, Be, Mu, Ba, Fr)$ with length $w = 235$.

The sixth iteration with distances $(0, 0, 0, 8, 0, 0, 0, 17, 20)$ yields $y = St$, $C = (Fr, Aa, Ha, Be, Mu, St, Ba, Fr)$ and $w = 247$.

For $k = 7$, we have $y = Nu$ and $C = (Fr, Aa, Ha, Be, Nu, Mu, St, BF, Fr)$ with length $w = 248$.

In the final iteration, $y = Du$. We obtain the tour

$$Fr \text{ — } Aa \text{ — } Du \text{ — } Ha \text{ — } Be \text{ — } Nu \text{ — } Mu \text{ — } St \text{ — } Ba \text{ — } Fr$$

with length $w = 250$. Thus FARIN has found – by sheer coincidence – the optimal tour constructed in Example 15.3.2 via the penalty approach; see Figure 15.6.

Exercise 15.5.4 (nearest insertion). Consider the procedure NEARIN which results from replacing step (4) of Algorithm 15.5.1 by

(4′) choose j with $d(j) = \min\{d(u) : u = 1, \ldots, n,\ u \notin C\}$.

Use this procedure to calculate a tour for the TSP of Example 15.1.2.

Several insertion algorithms are examined in [RoSL77]. More about heuristics, such as results concerning the quality in the metric case, empirical results, and probabilistic analysis, can be found in the books [LaLRS85] and [GuPa02].

15.6 Upper bounds: Local search

Having chosen a tour (at random or using a heuristic), the next step is to try to improve this tour as far as possible: we want to apply *post-optimization*. This means we consider sets of solutions that are *neighboring* in some sense and look for a *local optimum*. Let us formalize this idea.

Suppose $\mathbf{F}$ is the set of all feasible solutions for a given optimization problem; for example, for the TSP, $\mathbf{F}$ would be the set of all tours. A *neighborhood* is a mapping $N : \mathbf{F} \to 2^{\mathbf{F}}$: we say that N maps each $f \in \mathbf{F}$ to its *neighborhood* $N(f)$. Any algorithm which proceeds by determining local optima in neighborhoods is called a *local search algorithm*.

Lin [Lin65] proposed the following neighborhoods for a TSP on K_n with weight matrix $W = (w_{ij})$. Let f be a tour, and choose $k \in \{2, \ldots, n\}$. The neighborhood $N_k(f)$ is the set of all those tours g which can be obtained from f by first removing k arbitrary edges and then adding a suitable collection of k edges (not necessarily distinct from the removed edges). One calls $N_k(f)$ the *k-change neighborhood*. Any tour f which has minimal weight among all tours in $N_k(f)$ is said to be *k-optimal*. We can now describe a large family of local search algorithms for the TSP.

Algorithm 15.6.1 (k-opt). Let $W = (w_{ij})$ be the weight matrix of a TSP on K_n.

(1) Choose an initial tour f.
(2) **while** there exists $g \in N_k(f)$ with $w(g) < w(f)$ **do**
(3) choose $g \in N_k(f)$ with $w(g) < w(f)$;
(4) $f \leftarrow g$
(5) **od**

Of course, this generic algorithm leaves several choices unspecified. First, we have to decide how the initial tour should be chosen: at random or using one of the heuristics of Section 15.5. Also, in general there will be many possibilities for selecting a tour g in step (3). Two standard strategies are *first improvement* (we choose the first admissible g encountered) and *steepest descent* (we select a tour g of minimal weight in $N_k(f)$).

One could also run the algorithm several times, using distinct initial tours; in this case it makes sense to choose these tours randomly.

Perhaps the most important problem concerns which value of k one should choose. For large k (that is, larger neighborhoods), the algorithm yields a better approximation, but the complexity will grow correspondingly.[9] In practice, the value $k = 3$ proposed by [Lin65] seems to work well. We restrict ourselves to the simpler case $k = 2$ and examine it in more detail.

[9] Obviously, k-opt needs $O(n^k)$ steps for each iteration of the **while**-loop; nothing can be said about the number of iterations required.

Let f be a tour described by its edge set $f = \{e_1, \dots, e_n\}$:

$$x_1 \overset{e_1}{\text{---}} x_2 \overset{e_2}{\text{---}} \dots \overset{e_{n-1}}{\text{---}} x_n \overset{e_n}{\text{---}} x_1$$

is the corresponding Hamiltonian cycle. Then the tours $g \in N_2(f)$ can be found as follows: remove any two edges e_i and e_j from f, and connect the resulting two paths by inserting two edges e_i' and e_j'. We are interested only in the case $f \neq g$. Then e_i and e_j should not have a vertex in common, and this requirement determines e_i' and e_j' uniquely; see Figure 15.9.

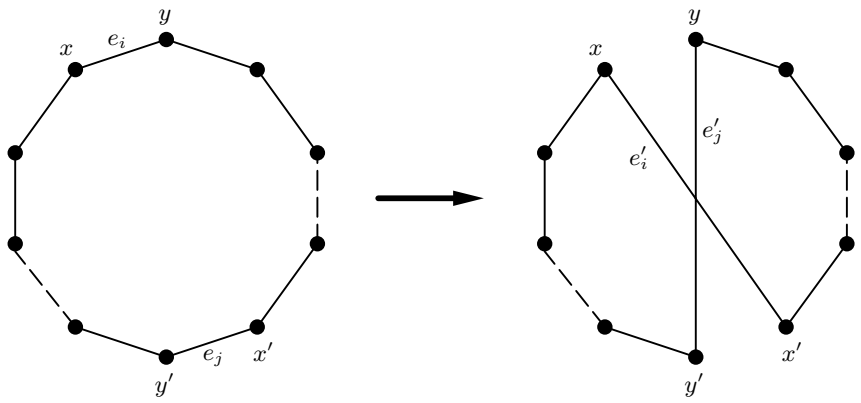


Fig. 15.9. A 2-exchange

Note that every neighborhood $N_2(f)$ contains precisely $n(n-3)/2$ tours $g \neq f$. For each such tour g, we put

$$\delta(g) \;=\; w(f) - w(g) \;=\; w(e_i) + w(e_j) - w(e_i') - w(e_j'). \tag{15.11}$$

Thus $\delta(g)$ measures the advantage which the tour g offers compared to f. We set $\delta = \max\{\delta(g) \colon g \in N_2(f)\}$; if $\delta > 0$, we replace f by some tour g with $\delta(g) = \delta$ (so we use steepest descent). Otherwise, f is already 2-optimal, and the algorithm 2-opt terminates. As noted before, each iteration has complexity $O(n^2)$; the number of iterations cannot be predicted. With these specifications, Algorithm 15.6.1 becomes the following algorithm proposed already in [Cro58].

Algorithm 15.6.2 (2-opt). Let $W = (w_{ij})$ be the weight matrix of a TSP on K_n, and let f be a tour with edge set $\{e_1, \dots, e_n\}$.

Procedure 2-OPT($W, f; f$)

(1) **repeat**
(2) $\quad\delta \leftarrow 0$, $g \leftarrow f$;
(3) $\quad$**for** $h \in N_2(f)$ **do**
(4) $\quad\quad$**if** $\delta(h) > \delta$ **then** $g \leftarrow h$; $\delta \leftarrow \delta(h)$ **fi**

(5) **od**
(6) $f \leftarrow g$;
(7) **until** $\delta = 0$

Note that Algorithm 15.6.2 has to terminate with a 2-optimal tour g: whenever step (4) is executed, the current tour is replaced by a better tour, so that the length of the tour decreases; obviously, this can happen only a finite number of times. It should be emphasized that the solution which the algorithm generates by no means has to be optimal: it is quite likely for the algorithm to get stuck in a bad neighborhood and produce only a local optimum. Therefore, it is common practice for 2-OPT (and for other local search algorithms) to run the algorithm repeatedly, starting with distinct initial tours.

Example 15.6.3. As usual, we consider the TSP of Example 15.1.2. Let us choose the tour of weight 266 constructed using Christofides' algorithm in Example 15.4.8 as our initial tour f; see Figure 15.8.

During the first iteration of 2-OPT, the edges $BeMu$ and $NuSt$ are replaced with $BeNu$ and $MuSt$; this yields the tour

$$Be — Nu — Mu — St — Ba — Fr — Du — Aa — Ha — Be$$

of length 253; see Figure 1.10.

The second iteration of 2-OPT exchanges the edges $FrDu$ and $AaHa$ for $FrAa$ and $DuHa$. The resulting tour is the optimal tour of length 250 shown both in Figure 15.6 and also in Figure 1.10. We have now constructed this tour in three different ways. Of course, it is coincidental that we reach an optimal solution by running a post-optimization procedure.

Exercise 15.6.4. Apply 2-OPT to the tour of Example 15.4.4; see Figure 15.7.

To speed up the running time, it might be a good idea to resort to the strategy *first improvement* and simply select the first tour g which is better than f in k-opt. In the special case of a metric TSP, it also makes sense not to consider all possible edge replacements, but to restrict the algorithm to edges being rather close to each other (according to the given metric).

A report about practical experiments with 2-OPT for large instances (up to a million cities) of the *Euclidean TSP*, where the distances are given by the Euclidean distance between n points in the plane, can be found in [Ben90].

Or [Or76] suggested a variation of 3-opt which examines only a small portion of all possible edge replacements, which cuts down the running time considerably, but nevertheless tends to yield good results. His basic idea is to try first to insert three consecutive cities of f between two other cities; if this improves the tour, the corresponding change is done immediately. If no more improvements can be achieved in this manner, the algorithm continues by considering pairs of consecutive cities, and so on.

An algorithm designed by Lin and Kernighan [LiKe73] has proved to be very efficient in practice; however, it is also considerably more involved. It uses variable values for k and decides during each iteration how many edges are to be replaced. The algorithm contains a number of tests, with the aim of checking – after r edges have been replaced already – whether it would make sense to exchange a further edge. Unfortunately, there are examples for which the Lin-Kernighan algorithm needs exponentially many steps; see [Pap92].

Nevertheless, the Lin-Kernighan heuristic and its variations – which have been studied extensively – provide without any doubt the most important approach to constructing good starting tours for large instances of the TSP. These methods tend to find nearly optimal tours within a reasonable amount of computation, even for extremely large problem instances. Two particularly important enhancements of the original Lin-Kernighan algorithm are due to Martin, Otto and Felten [MaOF91] and to Helsgaun [Hel00]. The interested reader may find a detailed account of this topic in [ApBCC06, Chapter 15].

More recently, a wealth of further heuristics of a rather different nature have been proposed; these are motivated by concepts from either physics or biology. We mention three important methods: *threshold accepting, tabu search*, and the *great deluge algorithm*. These methods sometimes yield results of surprising quality with relatively little effort; see, for example, [DuSc90], [Fie94], and [Due93]. The approach via genetic algorithms and evolution programs is interesting as well; see [Mic92, Chapter 10] and the references given there, in particular [MuGK81]. Unfortunately, it seems to be impossible to prove any theoretical results about the quality of such algorithms.

To sum up, the common approach to the TSP consists of several parts. First an initial tour is constructed using some heuristic (as in Section 15.5); usually, insertion algorithms are used for this task. Then a local search algorithm is applied to improve the current tour. Simultaneously, lower bounds are calculated (using the algorithm of Held and Karp of Section 15.3 or an LP-relaxation) to be able to judge the quality of the current solution.

Even for large instances of several thousand cities, it is nowadays usually possible to reduce the gap between the solution found and the optimal value to 1% or less. Not surprisingly, it is possible to construct degenerate examples for which the above techniques yield arbitrarily bad results [PaSt78], but examples coming from practical applications can generally be solved quite well. Finally, we also mention a general monograph on local search techniques: [AaLe97].

15.7 Exact neighborhoods and suboptimality

We saw in the previous section how neighborhoods are used for determining locally optimal solutions to hard problems. Using this approach, we hope that the locally optimal solution – or, perhaps, the best one among several local

optima – is pretty good also from a global point of view. Of course, the nicest thing that could possibly happen is that *every* local optimum is actually a global optimum. This suggests the following definition.

Let $\mathbf{F}$ be the set of admissible solutions of some optimization problem. A neighborhood $N : \mathbf{F} \to 2^{\mathbf{F}}$ is called an *exact neighborhood* if every locally optimal solution is already a global optimum.[10] Before examining exact neighborhoods for the TSP, we give an example for an (albeit polynomial) problem where exact neighborhoods are indeed helpful: the determination of minimal spanning trees.

Example 15.7.1. Let $G = (V, E)$ be a connected graph with weight function $w : E \to \mathbb{R}$. We define a neighborhood N as follows: for a given spanning tree T of G, $N(T)$ consists of precisely those spanning trees T' which result from T by adding some edge $e \notin T$ and then removing an arbitrary edge from the cycle $C_T(e)$; compare Section 4.3. The results proved there imply that this neighborhood is exact; to this end, it suffices to verify condition (4.1) in Theorem 4.3.1 for the global optimality of a given (locally optimal) tree T.

Assume that condition (4.1) is not satisfied. Then it is possible to find an edge $e \in E \setminus T$ and an edge $f \in C_T(e)$ such that $w(e) < w(f)$. Adding e and removing f then yields a tree $T' \in N(T)$ with $w(T') < w(T)$, which contradicts the local optimality of T. Note that this argument is just the first part of the proof of Theorem 4.3.1. Thus Theorem 4.3.1 may be viewed as a proof for the exactness of the neighborhood N; note that the correctness of the algorithms of Kruskal and Prim given in Section 4.4 basically rests on this fact.

Example 15.7.1 suggests the following approach to solving optimization problems:

(a) Find an exact neighborhood N.
(b) Find an efficient algorithm for examining the neighborhood $N(f)$ of a given feasible solution f: the algorithm should be able to recognize whether f is locally – and, hence, also globally – optimal; if this is not the case, it should replace f with a better solution f'.

Of course, it is not at all clear how efficient a search algorithm based on (a) and (b) would be: in general, it is not known how many neighborhoods $N(f)$ need to be examined until an optimum is found. Nevertheless, in view of our experiences with the TSP, we would probably be quite happy with an exact neighborhood and a polynomial algorithm for examining this neighborhood.[11]

[10] Of course, our previous experiences suggest that this idea will not be very helpful for the TSP; indeed, the present section will provide more bad news on the TSP in more than one respect. To use a phrase taken from [LaLRS85, p. 76]: *The outlook continues to be bleak.*

[11] It can be shown that not even the huge neighborhood N_{n-3} of Section 15.6 is exact.

Unfortunately, we will soon see that even this is only possible if we should have P = NP. This requires some effort. We begin by showing that the following problem is NP-complete.

Problem 15.7.2 (restricted Hamiltonian cycle, RHC). Let G be a graph, and let P be a Hamiltonian path of G. Does G contain a Hamiltonian cycle?

Even though this problem may seem simpler than HC, it is just as difficult: knowing the Hamiltonian path P does not help to decide if the graph G is Hamiltonian, as the following result due to Papadimitriou and Steiglitz [PaSt77] shows.

Theorem 15.7.3. *RHC is NP-complete.*

Proof. As HC is NP-complete by Theorem 2.7.4, it is sufficient to transform HC polynomially to RHC. To do this, we use the auxiliary graph D (D for *diamond*) shown in Figure 15.10. Let $G = (V, E)$ be a connected graph. We replace each vertex v of G by a diamond D_v; by adding suitable edges, we will construct a graph $G = (V', E')'$ with a Hamiltonian path; moreover, G' will contain a Hamiltonian cycle if and only if G does. As the number of vertices is multiplied only by 8, we will obtain a polynomial transformation of HC to RHC in this way.

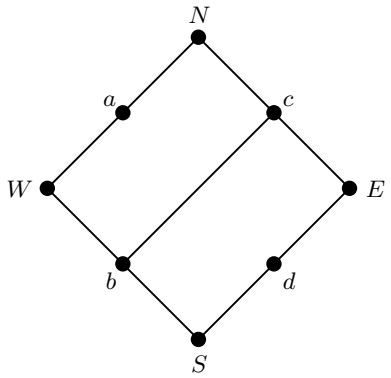
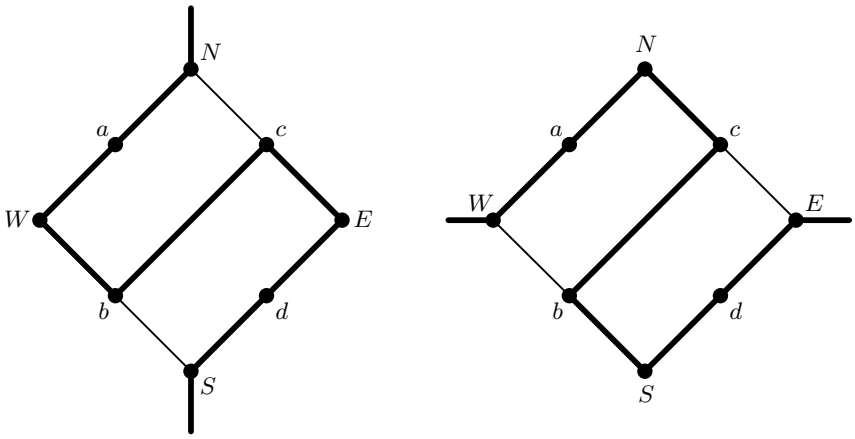


Fig. 15.10. A diamond D

When we specify E', we will impose the following restriction: all edges connecting a diamond D_v with $G' \setminus D_v$ have to be incident with one of the four vertices labelled as N, S, W, and E in Figure 15.10. As we shall show in a moment, this will guarantee that an arbitrary Hamiltonian cycle of G' can traverse a given diamond only in two ways: it has to contain one of the two paths shown in Figure 15.11. We will refer to these two possibilities as the *north-south method* and the *west-east method*, respectively.



north-south method west-east method

Fig. 15.11. Hamiltonian paths through D

It is actually rather easy to check our claim: suppose that C is a Hamiltonian cycle of G' which enters a diamond D via the vertex N. Then C has to contain the vertex a next; otherwise, a could not be contained in C at all, since $\deg a = 2$ and since N cannot be used a second time. Then the next vertex of C has to be W. Note that C cannot leave D in W: otherwise, C would have to enter D for a second time; but then it could not possibly contain all three vertices d, b, and c. Therefore C has to pass through b, c, E, d, and S in this order, leaving D from S. This indeed is just the north-south method. Similarly, a Hamiltonian cycle entering D in W has to traverse D according to the west-east method.

We now specify E', where we assume $V = \{1, \ldots, n\}$. First, we connect the n copies D_v of D (which form the vertex set of G') by adding the following $n-1$ edges: S_1N_2, S_2N_3, $\ldots$, $S_{n-1}N_n$. These edges create a Hamiltonian path path in G': the north-south method of D_1 followed by S_1N_2, followed by the north-south method of D_2, and so on to the edge $S_{n-1}N_n$ followed by the north-south method of D_n; see Figure 15.12.

In addition, we include the edges W_iE_j and W_jE_i in E', whenever ij is an edge of G. Then it is obvious that any Hamiltonian cycle C of G induces a Hamiltonian cycle C' in G': the diamonds in G' are visited in the same order as C visits the vertices of G, and each diamond is traversed using the west-east method.

Conversely, suppose that G' has a Hamiltonian cycle C'. Then C' cannot reach any of the diamonds D_i via N_i or S_i: otherwise, C' would have to pass through all of the diamonds using the north-south method. This would yield the Hamiltonian path W of G' shown in Figure 15.12, which cannot be

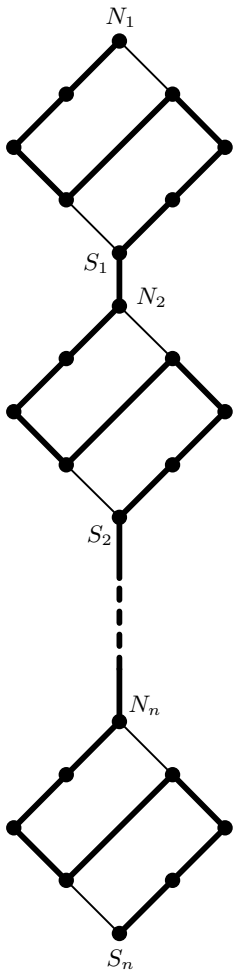


Fig. 15.12. A Hamiltonian path in G'

extended to a cycle since G' does not contain the edge N_1S_n. Therefore C' has to pass through all the diamonds by the west-east method, so that C' clearly induces a Hamiltonian cycle in G. □

We now use Theorem 15.7.3 to show that the following problem is likewise NP-complete, which is quite interesting in its own right: most likely we cannot even recognize an optimal tour when we see one. This result is also due to Papadimitriou and Steiglitz [PaSt77].

Problem 15.7.4 (TSP suboptimality). Suppose we are given a TSP and a tour. Is this tour *suboptimal*? That is, does there exist a shorter tour?

Theorem 15.7.5. *TSP suboptimality is NP-complete, even when restricted to the metric case.*

Proof. Clearly, the problem is in P, as any tour of shorter length constitutes a certificate for the answer *yes*. By Theorem 15.7.3, it will suffice to transform RHC to the problem in question. Let G be a graph on the vertex set $V = \{1, \ldots, n\}$, and let P be a Hamiltonian path for G. Consider the ΔTSP on K_n, where the weight matrix $W = (w_{ij})$ is defined as in the proof of Theorem 2.7.5, and note that P extends to a tour π of length

$$w(\pi) \;\leq\; (n-1) + 2 \;=\; n+1.$$

If π actually has length n, we have proved that G is Hamiltonian: tours of length n correspond to Hamiltonian cycles of G. Now assume $w(\pi) = n+1$. Then G contains a Hamiltonian cycle if and only if π is not optimal. By hypothesis, there is a polynomial algorithm deciding TSP suboptimality, so that the criterion just given can be checked efficiently. This shows that RHC would likewise be decidable in polynomial time. □

Theorem 15.7.6. *Assume the existence of an exact neighborhood N for the TSP and of a polynomial algorithm for deciding whether a given tour T is (locally and, hence, globally) optimal. Then $P = NP$.*

Proof. Note that such an algorithm would be able to decide TSP suboptimality in polynomial time. Thus the assertion follows from Theorem 15.7.5. □

Note that Theorem 15.7.6 asserts more than just the NP-completeness of TSP. Assume P $\neq$ NP. Then TSP is not solvable in polynomial time (by Theorem 2.7.5), so that no algorithm based on a polynomial local search process for an exact neighborhood could yield a solution after a polynomial number of iterations. Nevertheless, there still might be an exact neighborhood admitting a polynomial local search algorithm which needs, for instance, an exponential number of iterations. Theorem 15.7.6 excludes even this possibility.

Thus no neighborhood for the TSP which admits a polynomial local search algorithm can be exact (always assuming that P $\neq$ NP). In particular, this applies to the neighborhoods N_k (for fixed k), since N_k allows a local search algorithm of complexity $O(n^k)$. The following exercise strengthens this result by showing that a polynomial local search algorithm for a given neighborhood cannot lead to an ε-approximative algorithm for the TSP – even if we allow more than a polynomial number of iterations. Again, this result is due to Papadimitriou and Steiglitz [PaSt77].

Exercise 15.7.7. Let N be a neighborhood for the TSP admitting a polynomial algorithm **A** which checks whether a given tour is locally optimal and, if this is not the case, finds a better tour. Moreover, let $\mathbf{A}'$ be the local search algorithm based on **A** (as in Section 15.6). Suppose that $\mathbf{A}'$ is ε-approximative for some ε. Show that this implies P = NP (without any assumptions about the number of iterations).

Hint: Proceed by analogy to the proof of Theorem 15.4.1 and show that otherwise HC would be solvable in polynomial time. Why does $\mathbf{A}'$ need only a polynomial number of iterations for the instances of the TSP used in this argument?

Exercise 15.7.8. Prove that it is NP-complete to decide whether a given edge ij occurs in an optimal tour (for a given instance of the TSP).

Hint: Show that a polynomial algorithm for this decision problem would allow us to construct an optimal tour in polynomial time.

Exercise 15.7.9. In view of Exercises 15.7.7 and 15.7.8, it is tempting to try proving Theorem 15.7.5 by reducing the NP-completeness of TSP suboptimality to that of HC (and thus to avoid the detour via RHC). Discuss why this idea does not work.

Even though the considerations in this section show that we cannot expect any guarantee for the quality of the local search algorithms – such as 3-OPT and, in particular, Lin-Kernighan – which were discussed in Section 15.6, these algorithms nevertheless tend to work rather nicely in practice.

15.8 Optimal solutions: Branch and bound

This section is devoted to the problem of finding optimal solutions for the TSP. All the known techniques basically boil down to analyzing all possible solutions; this is not surprising because the TSP is an NP-complete problem. Nevertheless, it can make a huge difference in practice how one organizes this complete case analysis. Quite often, it is possible to find an optimal solution without too much effort by using some tricks (and hoping for a bit of good luck). We will illustrate this phenomenon via our standard example specified in 15.1.2.

The method we will use is called *branch and bound*; roughly, it works as follows. In each step, the set of all possible solutions is split into two or more subsets, which are represented by branches in a *decision tree*. For the TSP, an obvious criterion for dividing all tours into subsets is whether they contain a given edge or not. Of course, by itself this is just a useful way of organizing a complete case distinction. The approach becomes really useful only when we add an extra idea: in each step, we will calculate lower bounds for the weight of all solutions in the respective subtrees (using a suitable relaxation) and compare them with some previously computed upper bound (for instance, the length of a good tour found by a heuristic followed by post-optimization). Then no branch of the tree for which the lower bound exceeds the known upper bound can possibly lead to an optimal tour, so that this entire branch of the tree can be discarded; quite often, a large number of tours will be excluded in this way. Of course, the quality of this method will depend both on the

relaxation and on the branching criteria used; also, it can only be judged heuristically: we may not expect any theoretical performance guarantees.

We will present one of the oldest branch and bound techniques, which is due to Little, Murty, Sweeney and Karel [LiMSK63] (where the name *branch and bound* was introduced). Their algorithm was designed for the ATSP; of course, it may also be applied to the special case of the TSP. This algorithm is not particularly efficient, but it is easy to understand and easy to illustrate. As our purpose is merely to show how branch and bound algorithms work in principle, it will serve quite nicely. We refer the reader to [LaLRS85, Chapter 10] and [GuPa02, Chapter 4] for more recent branch and bound algorithms which use more advanced relaxations and more involved branching criteria.

Let $W = (w_{ij})$ be the weight matrix of a given TSP on K_n. We choose the diagonal entries w_{ii} as ∞ (as we did for the assignment relaxation); this can be interpreted as forbidding the use of loops. In order to calculate lower bounds, we shall transform the weight matrix in a manner similar to the transformation used in Section 15.3.

Let us select some row or column of W, and let us subtract a positive number d from all its entries, subject to the restriction that the resulting matrix W' should still have nonnegative entries only; thus we will use the smallest entry in our row or column. Recall that each tour π corresponds to a diagonal of the matrix W; hence it has to contain some entry of the row or column used for our transformation, so that the weight of π with respect to W' is reduced by d compared to its weight with respect to W. In particular, the optimal tours for W coincide with those for W'.

We continue this process until we obtain a matrix W'' having at least one entry 0 in each row and each column; such a matrix is called *reduced*. Note that the optimal tours for W agree with those for W'', and that the weight of each tour with respect to W'' is reduced by s compared to its weight with respect to W, where s is the sum of all the numbers subtracted during the reduction process. It follows that s is a lower bound for the weight of all tours (with respect to W). Note that the weight matrices resulting from the above construction will no longer be symmetric (in contrast to the transformations used in Section 15.3).

Let us illustrate the reduction process for the TSP of Example 15.1.2. We replace the diagonal entries 0 by ∞ and subtract, for each row of the matrix, the minimum of its entries; this yields the matrix $\tilde{W}$ displayed on the next page. Next we treat the columns of this matrix in the same manner and obtain the reduced matrix W'.

$\tilde{W}$:

	Aa	Ba	Be	Du	Fr	Ha	Mu	Nu	St
Aa	0	49	56	0	18	41	56	39	38
Ba	30	∞	61	27	7	56	10	16	0
Be	35	59	∞	28	27	0	31	15	34
Du	0	46	49	∞	15	35	55	36	33
Fr	6	14	36	3	∞	30	20	2	0
Ha	20	54	0	14	21	∞	51	34	41
Mu	47	20	43	46	23	63	∞	0	5
Nu	30	26	27	27	5	46	0	∞	2
St	27	8	44	22	1	51	3	0	∞

W' :

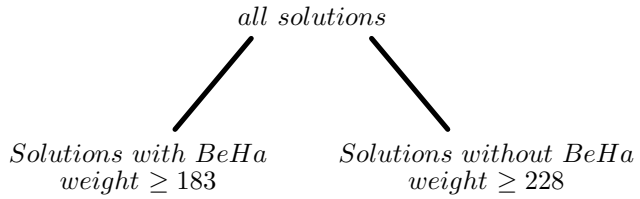
	Aa	Ba	Be	Du	Fr	Ha	Mu	Nu	St
Aa	∞	41	56	0	17	41	56	39	38
Ba	30	∞	61	27	6	56	10	16	0
Be	35	51	∞	28	26	0	31	15	34
Du	0	38	49	∞	14	35	55	36	33
Fr	6	6	36	3	∞	30	20	2	0
Ha	20	46	0	14	20	∞	51	34	41
Mu	47	12	43	46	22	63	∞	0	5
Nu	30	18	27	27	4	46	0	∞	2
St	27	0	44	22	0	51	3	0	∞

The sum of all the numbers subtracted is

$$s \;=\; 8+27+29+8+20+29+17+17+19+8+1 \;=\; 183,$$

so that each tour has weight at least 183 with respect to W. Of course, we found better bounds earlier: for example, 230 using the s-tree relaxation in Example 15.2.4, but this does not help us in the present context.

Next, we have to choose an edge ij to split the set of solutions; note that we have to use directed edges ij here. Tours not containing ij can then be described by the weight matrix M which results from W' by replacing the (i,j)-entry by ∞. As we would like to increase the current lower bound s, we should choose some edge that corresponds to a zero entry in W' and, moreover, is the only 0-entry in its row and column (so that the matrix M will allow a further reduction). Clearly, it makes sense to choose the entry 0 for which M can be reduced by the largest possible amount. In our example, this is the edge $BeHa$, which leads to a reduction of $30+15=45$ for M. Thus the first part of the decision tree looks as follows.



Recall that we already know an optimal tour (of weight 250), which we can obtain from the FARIN heuristic; see Example 15.5.3 and Figure 14.6. Our present considerations should, of course, confirm that that tour is indeed optimal. Still, that tour does not help us to exclude one of the branches of the decision tree yet. Of course, such limited progress is to be expected, because the known solution of weight 250 contains the edge $HaBe$ (for an appropriate orientation) and thus occurs on the right branch of the decision tree. As our original TSP was symmetric, it actually suffices to consider this branch: for each tour containing the edge $BeHa$, there is a corresponding tour of the same weight not containing this edge, namely the tour having the opposite orientation. Thus we may replace W' by the following weight matrix M.

$$M: \quad \begin{array}{c} \\ Aa \\ Ba \\ Be \\ Du \\ Fr \\ Ha \\ Mu \\ Nu \\ St \end{array} \begin{array}{c} \begin{array}{ccccccccc} Aa & Ba & Be & Du & Fr & Ha & Mu & Nu & St \end{array} \\ \begin{pmatrix} \infty & 41 & 56 & 0 & 17 & 11 & 56 & 39 & 38 \\ 30 & \infty & 61 & 27 & 6 & 26 & 10 & 16 & 0 \\ 20 & 36 & \infty & 13 & 11 & \infty & 16 & 0 & 19 \\ 0 & 38 & 49 & \infty & 14 & 5 & 55 & 36 & 33 \\ 6 & 6 & 36 & 3 & \infty & 0 & 20 & 2 & 0 \\ 20 & 46 & 0 & 14 & 20 & \infty & 51 & 34 & 41 \\ 47 & 12 & 43 & 46 & 22 & 33 & \infty & 0 & 5 \\ 30 & 18 & 27 & 27 & 4 & 16 & 0 & \infty & 2 \\ 27 & 0 & 44 & 22 & 0 & 21 & 3 & 0 & \infty \end{pmatrix} \end{array}$$

Again, we use that entry 0 in M which allows the largest possible further reduction when we replaced it with ∞ in order to split up the set of possible solutions: (Ha, Be), where the possible reduction is $14 + 27 = 41$. Then the part of the decision tree which belongs to tours which contain neither $HaBe$ nor $BeHa$ has a lower bound of 269; hence this branch can be discarded in view of the known tour of weight 250. In other words: every optimal tour for the TSP of Example 15.1.2 has to contain the edge $HaBe$.[12] As none of the tours which we still need to consider may contain a further edge beginning in

[12] This fact is not all that surprising, as $HaBe$ is the shortest edge incident with Be, while all the other edges are considerably longer.

Ha or ending in Be, we may omit both the row Ha and the column Be from M. Next we use the (Aa, Du)-entry of the resulting 8×8-matrix. For tours not containing the edge $AaDu$, we may replace this entry by ∞ and reduce the resulting matrix by $11 + 3 = 14$. This yields the following matrix A, with the associated lower bound 242.

$$A: \quad \begin{array}{c} \\ Aa \\ Ba \\ Be \\ Du \\ Fr \\ Mu \\ Nu \\ St \end{array} \begin{array}{c} \begin{array}{cccccccc} Aa & Ba & Du & Fr & Ha & Mu & Nu & St \end{array} \\ \begin{pmatrix} \infty & 30 & \infty & 6 & 0 & 45 & 28 & 27 \\ 30 & \infty & 24 & 6 & 26 & 10 & 16 & 0 \\ 20 & 36 & 10 & 11 & \infty & 16 & 0 & 19 \\ 0 & 38 & \infty & 14 & 5 & 55 & 36 & 33 \\ 6 & 6 & 0 & \infty & 0 & 20 & 2 & 0 \\ 47 & 12 & 43 & 22 & 33 & \infty & 0 & 5 \\ 30 & 18 & 24 & 4 & 16 & 0 & \infty & 2 \\ 27 & 0 & 19 & 0 & 21 & 3 & 0 & \infty \end{pmatrix} \end{array}$$

We shall investigate A later. First, we examine those tours which contain the edge $AaDu$. For such tours, both the row Aa and the column Du may be omitted from M. Moreover, the edge $DuAa$ cannot also occur, so that the corresponding entry may be replaced by ∞. Then the resulting matrix can be reduced even further: we may subtract 6 from the column Aa and 5 from the row Du. This yields the following matrix M' with associated lower bound 239.

$$M': \quad \begin{array}{c} \\ Ba \\ Be \\ Du \\ Fr \\ Mu \\ Nu \\ St \end{array} \begin{array}{c} \begin{array}{ccccccc} Aa & Ba & Fr & Ha & Mu & Nu & St \end{array} \\ \begin{pmatrix} 24 & \infty & 6 & 26 & 10 & 16 & 0 \\ 14 & 36 & 11 & \infty & 16 & 0 & 19 \\ \infty & 33 & 9 & 0 & 50 & 31 & 28 \\ 0 & 6 & \infty & 0 & 20 & 2 & 0 \\ 41 & 12 & 22 & 33 & \infty & 0 & 5 \\ 24 & 18 & 4 & 16 & 0 & \infty & 2 \\ 21 & 0 & 0 & 21 & 3 & 0 & \infty \end{pmatrix} \end{array}$$

Let us consider tours not containing the edge $FrAa$ next. Then the entry corresponding to $FrAa$ may be replaced by ∞, so that the matrix can be reduced by 14. That yields a new lower bound of 253, so that this branch of the decision tree can again be discarded (in view of the known tour of weight 250). Thus we may restrict our attention to tours containing the edge $FrAa$. We can now omit both the row Fr and the column Aa from M'. As our tour contains the edges $FrAa$ and $AaDu$, the edge $DuFr$ is no longer permissible, so that we replace the corresponding entry by ∞; note that this does not lead

to any further possibilities for reducing the matrix, so that the lower bound 239 stays unchanged. The resulting matrix M'' looks as follows.

$$M'': \quad \begin{array}{c} \\ Ba \\ Be \\ Du \\ Mu \\ Nu \\ St \end{array} \begin{array}{c} \begin{array}{cccccc} Ba & Fr & Ha & Mu & Nu & St \end{array} \\ \begin{pmatrix} \infty & 6 & 26 & 10 & 16 & 0 \\ 36 & 11 & \infty & 16 & 0 & 19 \\ 33 & \infty & 0 & 50 & 31 & 28 \\ 12 & 22 & 33 & \infty & 0 & 5 \\ 18 & 4 & 16 & 0 & \infty & 2 \\ 0 & 0 & 21 & 3 & 0 & \infty \end{pmatrix} \end{array}$$

Next we consider tours which do not involve the edge $DuHa$; this yields a lower bound of 283, so that we may restrict our attention to tours containing $DuHa$. Thus we discard both the row Du and the column Ha. Moreover, the tours still left all contain the path (Fr, Aa, Du, Ha, Be), which precludes using the edge $BeFr$; hence the corresponding entry is replaced by ∞. In the next step, we find that the tour has to contain the edge $BeNu$: without this edge, we get a lower bound of $239 + 16 = 255$, which once again exceeds the known upper bound. Then we must also discard the edge $NuFr$; replacing the corresponding entry by ∞ allows us to subtract 5 from row Mu. Now our lower bound has increased to 244, and we are left with the following matrix.

$$\begin{array}{c} \\ Ba \\ Mu \\ Nu \\ St \end{array} \begin{array}{c} \begin{array}{cccc} Ba & Fr & Mu & St \end{array} \\ \begin{pmatrix} \infty & 6 & 10 & 0 \\ 7 & 17 & \infty & 0 \\ 18 & \infty & 0 & 2 \\ 0 & 0 & 3 & \infty \end{pmatrix} \end{array}$$

We may now insert the edge $MuSt$ into our tour: for tours not containing this edge, we obtain a lower bound of 251. Next we omit both the row Mu and the column St and replace the (St, Mu)-entry by ∞. Subtracting 6 from row Ba, we get a lower bound of 250, and this means we may stop: the known tour of this length is at least as good as any tour in the branch under investigation.

Continuing the procedure in the same way would yield the additional edges $StBa$, $BaFr$, and $NuMu$, so that we would obtain the (optimal) tour

$$Fr \text{ — } Aa \text{ — } Du \text{ — } Ha \text{ — } Be \text{ — } Nu \text{ — } Mu \text{ — } St \text{ — } Ba \text{ — } Fr$$

of length 250 which we already know. Thus we could have obtained this tour without using heuristic methods by performing a sort of DFS on the decision tree: always choose the branch with the smallest lower bound for continuing the investigation. Of course, it is then possible that we discover only at a later

point of the decision tree that we could have discarded some of the earlier branches.

It remains to investigate the branch of the decision tree which belongs to the matrix A on page 493. Suppose that a tour in this branch does not contain the edge $DuAa$. Then the matrix can be reduced by $5 + 6$, which yields a lower bound of 253, so that the corresponding branch can be discarded. Hence we insert the edge $DuAa$ into our tour, and discard both the row Du and the column Aa from A. Next, we see that an optimal tour on this branch of the tree has to contain $BeNu$: otherwise, we obtain a lower bound of 252. Hence we may omit also the row Be and the column Nu; moreover, we have to replace the (Ha, Nu)-entry by ∞, since $HaBe$ and $BeNu$ are contained in the tour. Then we can subtract 5 from row Mu, which yields a lower bound of 247 and the matrix A' below.

$$A' : \begin{array}{c} \\ Aa \\ Ba \\ Fr \\ Mu \\ Nu \\ St \end{array} \begin{array}{c} \begin{array}{cccccc} Ba & Du & Fr & Ha & Mu & St \end{array} \\ \begin{pmatrix} 30 & \infty & 6 & 0 & 45 & 27 \\ \infty & 24 & 6 & 26 & 10 & 0 \\ 6 & 0 & \infty & 0 & 20 & 0 \\ 7 & 38 & 17 & 28 & \infty & 0 \\ 18 & 24 & 4 & \infty & 0 & 2 \\ 0 & 19 & 0 & 21 & 3 & \infty \end{pmatrix} \end{array}$$

Next we insert the edge $FrDu$, as leaving out this edge would increase the lower bound by 19 to 266; omit the corresponding row and column; and replace the (Aa, Fr)-entry by ∞. This forces us to include the edge (Aa, Ha): otherwise we would increase the lower bound to 295. Now we can replace the (Nu, Fr)-entry by ∞. Then the edge (Mu, St) has to be contained in the tour, and the (St, Mu)-entry is changed to ∞. After having omitted the appropriate rows and columns, we are left with a (3×3)-matrix which can be reduced by 6 in row Ba. This yields a new lower bound of 253, so that we can actually ignore all tours on the branch of the decision tree belonging to A, and the algorithm terminates. By complete analysis of all possibilities, we have now proved that our tour of length 250 is optimal. Figure 15.13 shows the whole decision tree.

As noted before, the partial tour in the left-most branch can be completed in a unique manner, which yields the known optimal tour of weight 250. Also note that the branch with lower bound 253 indeed contains a tour of this weight, which may be obtained by exchanging the order of Aa and Du in the optimal tour. Similarly, the branch with lower bound 251 contains a tour of this length, which results from exchanging the order of Ba and St in the optimal tour.

We refrain from stating the preceding branch and bound algorithm formally; the worked example should suffice to illustrate how the algorithm

works. An explicit formulation as well as a PASCAL program and an ATSP-example of size 6 can be found in [SyDK83].

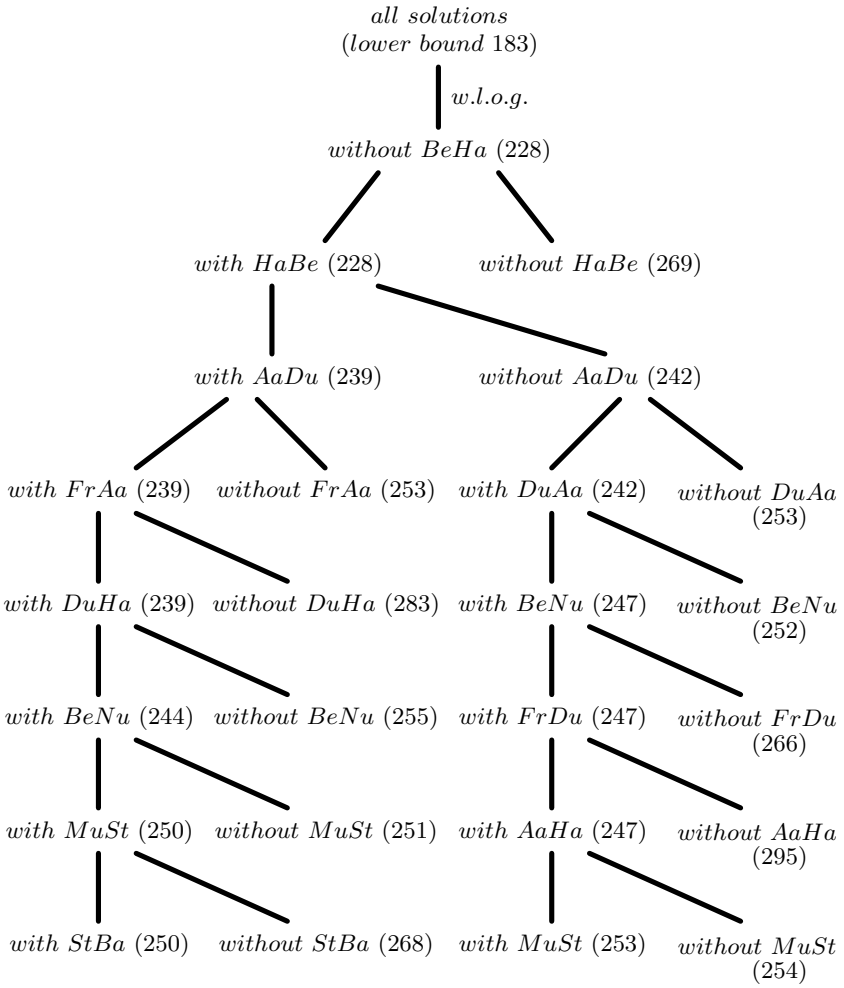


Fig. 15.13. Decision tree for the TSP of Example 15.1.2

We close this section with some remarks concerning the preceding algorithm. The numbers which were subtracted from the rows and columns of the original weight matrix W define two vectors: $\mathbf{u} = (8, 27, 29, 8, 20, 29, 17, 17, 19)$ and $\mathbf{v} = (0, 8, 0, 0, 1, 0, 0, 0, 0)$. The reduction process does not only change the weight of every tour by a constant (in this case, 183): it actually changes the weight of each diagonal of the matrix, that is, of each matching for the

corresponding assignment problem. Hence the bound of $\sum_i (u_i + v_i) = 183$ is actually a bound for the corresponding assignment relaxation; compare Section 15.2.

This is not really surprising: using the terminology of Section 14.2, $(\mathbf{u}, \mathbf{v})$ is a feasible node weighting for the corresponding assignment problem, albeit in its minimization version, which can be treated in analogy to Chapter 14; indeed, always $u_i + v_j \leq w_{ij}$. Thus the algorithm of [LiMSK63] is based on relaxing the assignment relaxation of the TSP even further, which leads to rather weak lower bounds.

One final comment: replacing certain entries by ∞ – that is, forbidding the corresponding edges – during the algorithm corresponds to excluding non-cyclic permutations. If one would omit this feature from the algorithm, it would determine a complete matching of minimal weight instead of an optimal tour.

15.9 Concluding remarks

As we mentioned in Section 15.2, TSP's are often solved by linear programming methods, using the polytope P associated with the tours. Recall that two vertices of a polytope are called *adjacent* if they are incident with a common edge. It can be shown that the unique minimal exact neighborhood for a wide class of optimization problems is always given by adjacency in the corresponding polytope; see [PaSt82], §19.7.

In particular, for the TSP, the minimal exact neighborhood $N(f)$ of a tour f contains precisely those tours which are adjacent to f in P. Unfortunately, this does not help us either: the question whether two vertices of P are adjacent is again an NP-complete problem; see [Pap78]. In this context, we mention a truly amazing result about the polytope P' corresponding to the ATSP: the diameter of P' (in the graph theoretical sense, where we consider the vertices and edges of P' as a graph called the *skeleton* of P') is 2; see [PaRa74]. Thus the distance in P' between an arbitrary tour and a given optimal tour is just 2! We refer the reader to [LaLRS85, Chapter 8] and to [GuPa02] for details – and references – concerning the polytopes associated with the TSP.

We now turn to some algorithmic consequences of the polyhedral approach. As mentioned before, even large instances of the TSP – consisting of several thousands (or even more) cities – can nowadays be approximated very well: it is routinely possible to find a solution which is not more than 1% worse than the optimal solution.

But what about determining optimal solutions? The first larger instance of a TSP for which an optimal solution was found appears in the seminal paper by Dantzig, Fulkerson and Johnson [DaFJ54]; this instance involved 49 cities: Washington, D.C. and the 48 capitals of the then 48 states of the USA.

These authors first determined a good solution heuristically, which was actually already optimal, and then used the LP relaxation together with adding appropriate inequalities; compare Section 15.2. Their paper is considered a milestone in the history of combinatorial optimization; it is the forerunner of virtually all algorithms used at present for solving TSPs to optimality. The main part of these algorithms is the LP relaxation, which can be summarized as follows.

(1) Choose some LP relaxation for the TSP, and determine an optimal solution $\mathbf{x}$ for this relaxation.

(2) If $\mathbf{x}$ is not a tour, try to find one or more valid inequalities for the polytope P which are violated by $\mathbf{x}$. Add these inequalities to the LP, and replace $\mathbf{x}$ by a solution of the new LP. If possible, repeat this step until there are no longer any violated inequalities.[13]

Using just this method, Grötschel [Gro80] could solve a problem involving 120 German cities (where the distances were taken from the *Deutscher Generalatlas*) in merely 13 iterations – without using a branch and bound technique. The results of [PaHo80] and [CrPa80] indicate that LP relaxation usually yields very good lower bounds, if not actually an optimal solution. Thus, even if the LP relaxation does not produce an optimum, it should at least establish that some heuristically constructed tour is a really good approximate solution. Then it is also quite often possible to construct an optimal solution, using branch and bound; solving problems involving a few hundred cities is nowadays more or less routine. In [CrPa80], a problem with 318 cities was solved via this approach; this was the world record for several years. Since then, this record has been surpassed quite a few times; see Table 15.1 which contains some truly amazing results.

We now give a brief discussion of the first problem listed in Table 15.1 (with 532 cities), in order to emphasize the importance of efficient heuristics for solving large TSPs to optimality via the LP relaxation method. As the graph K_{532} contains precisely $141,246$ edges, the LP treated by [PaRi87] has just this number of (structural) variables (and some additional slackness variables). Now Padberg and Rinaldi applied the Lin-Kernighan heuristic mentioned earlier to 50 initial tours chosen at random; the resulting locally optimal tours contained altogether only 1278 edges. The corresponding 1278 variables were used for the first LP relaxation: all other variables were fixed at 0. Throughout the whole computation for solving this (already rather large) problem, no LP occurred which involved more than 1520 structural variables or more than 815 restrictions.

If one settles for extremely good approximations, one may reach considerably larger orders of magnitude. We mention a particularly impressive example: the World TSP tour of length 7,516,024,785 found by Keld Helsgaun in

[13] Of course, it will usually be necessary to abort this process at some point: there is no guarantee at all that it has to terminate.

Table 15.1. Some large TSPs solved to optimality

Year	Cities	Reference
1987	532	[PaRi87]
1991	666	[GrHo91]
1991	1,002	[PaRi91]
1991	2,392	[PaRi91]
1995	7,397	[ApBCC95]
1998	13,509	[ApBCC98]
2001	15,112	[ApBCC01]
2004	18,512	[ApBCC06]
2004	24,978	[ApBCC06]
2004	33,810	[ApBCC06]
2006	85,900	[ApBCC06]

February 2007, using an improved version of his Lin-Kernighan code originally presented in [Hel00]. The best present lower bound stands at 7,512,082,035; this bound was obtained – also in February 2007 – by Applegate, Bixby, Chvátal, and Cook using their Concorde TSP code [ApBCC04]. Thus Helsgaun's tour is at most 0.0525% longer than an optimal tour through the given 1,904,711 cities.

The facts just reported come from the excellent website by Applegate, Bixby, Chvátal, and Cook concerning the TSP:

`http://www.tsp.gatech.edu`

and update the information given in their recent book [ApBCC06], which is highly recommended – it is the definitive study of computational aspects of the TSP, and is likely to remain so for quite some time.

For a more detailed description of the LP relaxation method, which was merely sketched here, we refer the reader also to the following older sources: [LaLRS85, Chapter 9], [PaRi91], and [GuPa02]. This technique is a special case of a general method which proved to be efficient for several other hard optimization problems as well. It belongs to the field of *polyhedral combinatorics*, an important part of combinatorial optimization. We refer the reader to the interesting surveys of [Pul83] and [Gro84], and to the monumental work [Schr03]. We also recommend the book [GrLS93] and [Rei94], an earlier monograph on computational aspects of the TSP and its applications.

Finally, let us mention that several other NP-hard problems have been attacked with a remarkable degree of success. For instance, maximal cliques in graphs with up to 400 vertices and $30,000$ edges have been found; see [BaYu86]. Similar results for the corresponding weighted problem are in [BaXu91].

A

Some NP-Complete Problems

To ask the hard question is simple.
But what does it mean?
What are we going to do?
W. H. Auden

In this appendix we present a brief list of NP-complete problems; we restrict ourselves to problems which either were mentioned before or are closely related to subjects treated in the book. A much more extensive list can be found in Garey and Johnson [GaJo79].

Chinese postman (cf. Section 14.5)

Let $G = (V, A, E)$ be a mixed graph, where A is the set of directed edges and E the set of undirected edges of G. Moreover, let w be a nonnegative length function on $A \cup E$, and c be a positive number. Does there exist a cycle of length at most c in G which contains each edge at least once and which uses the edges in A according to their given orientation?

This problem was shown to be NP-complete by Papadimitriou [Pap76], even when G is a planar graph with maximal degree 3 and $w(e) = 1$ for all edges e. However, it is polynomial for graphs and digraphs; that is, if either $A = \emptyset$ or $E = \emptyset$. See Theorem 14.5.4 and Exercise 14.5.6.

Chromatic index (cf. Section 9.3)

Let G be a graph. Is it possible to color the edges of G with k colors, that is, does $\chi'(G) \leq k$ hold?

Holyer [Hol81] proved that this problem is NP-complete for each $k \geq 3$; this holds even for the special case where $k = 3$ and G is 3-regular.

Chromatic number (cf. Section 9.1)

Let G be a graph. Is it possible to color the vertices of G with k colors, that is, does $\chi(G) \leq k$ hold?

Karp [Kar72] proved that this problem is NP-complete for each $k \geq 3$. Even the special case where $k = 3$ and G is a planar graph with maximal

degree 4 remains NP-complete; see [GaJS76]. Assuming P $\neq$ NP, there is not even a polynomial approximative algorithm which always needs fewer than $2\chi(G)$ colors; see [GaJo76]. For perfect graphs, the chromatic number can be computed in polynomial time; see [GrLS93].

Clique (cf. Exercise 2.8.3)

Let $G = (V, E)$ be a graph and $c \leq |V|$ a positive integer. Does G contain a clique consisting of c vertices?

This problem is NP-complete by a result of Karp [Kar72]; thus determining the clique number $\omega(G)$ is an NP-hard problem. Also, the related question of whether G contains a clique with at least $r|V|$ vertices, where $0 < r < 1$, is NP-complete for fixed r. Assuming P $\neq$ NP, there is not even a polynomial ε-approximative algorithm for determining a maximal clique; see [ArSa02]. However, the problem can be solved in polynomial time for perfect (in particular, for bipartite) graphs; see [GrLS93].

Diameter (cf. Section 3.9)

Let $G = (V, E)$ be a connected graph, and let $c \leq |V|$ be a positive integer. Recall that the diameter of G can be determined efficiently; see Section 3.9. However, the following two related problems are NP-complete; see [ChTh78] and [GaJo79].

(1) Does there exist a strongly connected orientation H of G with diameter at most c? Note that Robbins' theorem allows us to check efficiently whether a strongly connected orientation exists (and to find such an orientation, if possible); see Section 1.6.
(2) Let C be a given set of at most $|E|$ nonnegative integers. Does there exist a mapping $w\colon E \to C$ such that G has *weighted diameter* at most c, that is, such that any two vertices u, v have distance $d(u, v) \leq c$ in the network (G, w)? This problem remains NP-complete even for $C = \{0, 1\}$.

Discrete metric realization (cf. Section 3.2)

Let $D = (d_{xy})$ be an $n \times n$ matrix with integer entries representing distances in a finite metric space . Is there a network (G, w) of total length $\leq k$ realizing D?

Winkler [Win88] proved that this problem – and also the the analogous real problem – is NP-complete.

Disjoint paths (cf. Section 7.1)

Let $G = (V, E)$ be a graph, s and t be two vertices of G, and k and c be two

positive integers. Does G contain k vertex disjoint paths of length at most c from s to t?

Itai, Pearl and Shiloach [ItPS82] proved that this problem is NP-complete for each fixed $k \geq 5$ (whereas it is polynomial for fixed $k \leq 4$). Similar results hold for edge disjoint paths from s to t, and for the analogous problems where each path should contain precisely c edges. In contrast, the maximal number of (edge or vertex) disjoint paths from s to t can be determined efficiently using network flow methods if no restrictions are added; see Section 7.1.

Graph partitioning (cf. Section 9.2)

Let $G = (V, E)$ be a graph and c a positive integer. The question of whether G can be partitioned into at most c subgraphs of a given type is NP-complete for many classes of subgraphs: for triangles and, more generally, for subgraphs with a given isomorphism type, for Hamiltonian subgraphs, for forests, for cliques, and for matchings. We refer the reader to [GaJo79, §A1.1] and the references given there.

In particular, determining the clique partition number $\theta(G)$ is an NP-hard problem in general. For perfect graphs, this problem can be solved in polynomial time; see [GrLS93].

Hamiltonian cycle (cf. Sections 1.4 and 2.8)

Let $G = (V, E)$ be a graph. Does G contain a Hamiltonian cycle?

Karp [Kar72] proved that this problem is NP-complete; it remains NP-complete even if we know a Hamiltonian path of G [PaSt77]; see Theorem 15.7.3. The special cases for bipartite graphs and for planar, 3-connected, 3-regular graphs are still NP-complete; see [Kri75] and [GaJT76]. The analogous problem for directed Hamiltonian cycles in digraphs likewise is NP-complete [Kar72]; see Exercise 2.7.6.

Hamiltonian path (cf. Exercise 2.7.7)

Does the graph $G = (V, E)$ contain a Hamiltonian path? This problem and the analogous directed problem are NP-complete, even if the start and end vertices of the Hamiltonian path are fixed; see [GaJo79].

Independent set (cf. Exercise 2.8.3)

Let $G = (V, E)$ be a graph and $c \leq |V|$ a positive integer. Does G contain an independent set with c elements? Note that the independent sets of G are precisely the cliques of the complementary graph $\overline{G}$. This problem is therefore NP-complete in general, but polynomial for perfect graphs (see **Clique** and **Vertex cover**).

The independent set problem remains NP-complete when restricted to 3-regular planar graphs; see [GaJS76].

Induced subgraph

Let $G = (V, E)$ be a graph and c a positive integer. The problem of whether G contains an induced subgraph on c vertices that belongs to a prescribed class of graphs is often NP-complete: for cliques and independent sets (see **Clique** and **Independent set**), and also for planar subgraphs, for bipartite subgraphs, for subforests, etc. We refer to [GaJo79, §A1.2] and the references given there.

Integer linear programming (cf. Section 14.3)

Let A be an $m \times n$ integer matrix, $\mathbf{c} \in \mathbb{Z}^n$ and $\mathbf{b} \in \mathbb{Z}^m$ integer vectors, and d an integer. Does there exist an integer vector $\mathbf{x} \in \mathbb{Z}^n$ satisfying $\mathbf{x} \geq \mathbf{0}$, $A\mathbf{x}^T \leq \mathbf{b}^T$, and $\mathbf{c}\mathbf{x}^T \geq d$? This problem is NP-complete by a result of Karp [Kar72], whereas the corresponding linear program (where $\mathbf{x}$ may have rational entries) can be solved in polynomial time by the work of Khachyan [Kha79].

Longest cycle

Let $N = (G, w)$ be a network with a nonnegative length function w, where G is a graph, and let c be a positive integer. Does N contain a cycle of length at least c?

This problem is NP-complete even when all edges have length 1; see Exercise 2.7.8. Similar results hold for the analogous directed problem.

Longest path (cf. Sections 2.7 and 3.1)

Let s and t be two vertices in a network $N = (G, w)$ on a graph G, where w is a nonnegative length function, and let c be a positive integer. Does there exist a path from s to t of length at least c?

This problem is NP-complete even when all edges have length 1; see Exercise 2.7.8. Similar results hold for the analogous directed problem.

Matroid intersection (cf. Section 5.4)

Let $(E, \mathbf{S}_i)$ $(i = 1, 2, 3)$ be three matroids on the same set E, and let c be a positive integer. Does E have a subset U of cardinality c which is an independent set for all three matroids?

This problem is NP-complete; see Theorem 5.4.13. Note that the corresponding problem for the intersection of two matroids is solvable in polynomial time (even in the weighted case); see [Law75, Law76], [Edm79], [Cun86], and [Whi87].

Max cut (cf. Chapter 6)

Let $G = (V, E)$ be a graph with a nonnegative capacity function c, and let b be a positive integer. Does there exist a cut (S, T) of G with capacity $c(S, T) \geq b$?

This problem is NP-complete by a result of Karp [Kar72]; this holds even in the special case where G has maximal degree 3 and $c(e) = 1$ for all edges e; see [Yan78]. Thus determining a cut of maximal capacity is an NP-hard problem, whereas the analogous problem for cuts of minimal capacity is easy.

However, the max cut problem is polynomial for planar graphs; see [Had75].

Min cut (cf. Chapter 6)

Let $G = (V, E)$ be a graph with a nonnegative capacity function c, let s and t be two vertices of G, and $b \leq |V|$ and k be two positive numbers. Does there exist a cut (S, T) of G with $s \in S$, $t \in T$, $|S| \leq b$, $|T| \leq b$, and capacity $c(S, T) \leq k$?

This problem is NP-complete, even when $c(e) = 1$ for all edges e; see [GaJS76]. Note that omitting the bounds on $|S|$ and $|T|$ (that is, putting $b = |V|$) yields one of the fundamental easy problems: again, we have a case of an easy problem becoming hard due to additional constraints.

Minimum k-connected subgraph (cf. Chapter 8)

Let $G = (V, E)$ be a graph, and let $k \leq |V|$ and $b \leq |E|$ be two positive integers. Does there exist a subset E^* of E with $|E^*| \leq b$ such that $G^* = (V, E^*)$ is k-connected?

This problem – and also the analogous problem for k-fold line connectivity – is NP-complete for each fixed $k \geq 2$; see [GaJo76]. Thus determining a minimal k-connected subgraph of G is NP-hard. Note that the case $k = 1$ can be solved with complexity $O(|E|)$ using BFS, for example: then we merely have to find a spanning tree of G.

Minimum spanning tree (cf. Chapter 4)

Let $N = (G, w)$ be a network with a nonnegative weight function w on a connected graph G. As we saw in Section 4.3, determining a minimal spanning tree T is one of the fundamental easy problems of algorithmic graph theory. As for the problem of determining spanning trees in general, we obtain NP-complete problems by adding side constraints, for example by restricting the diameter of T. See Section 4.7 and [GaJo79, §A.2.1].

Network flow (cf. Chapters 6 and 10)

The flow problems we treated in this book are all solvable in polynomial time.

Again, adding side constraints will often result in NP-complete problems. We refer the reader to [GaJo79, §A.2.4] and the references given there.

Network reliability (cf. Example 3.1.2)

Let $G = (V, E)$ be a graph, V^* a subset of V, p a mapping from E to the rational numbers in $[0, 1]$ (the *failure probability*), and $q \leq 1$ a positive rational number. Is the probability that any two vertices in V^* are connected by at least one *reliable* path (that is, a path which does not contain an edge which fails) at least q?

This problem is NP-complete by a result of Rosenthal [Ros77]; see also [Val79b] for related questions. Provan [Pro86] showed that it is NP-hard to determine the probability for the existence of a reliable path from s to t in a planar acyclic digraph G, and also in a planar graph G with maximal degree $\Delta(G) = 3$.

Permanent evaluation (cf. Section 7.4)

Let A be an $n \times n$ matrix with entries 0 and 1, and let $k \leq n!$ be a positive integer. Does $\operatorname{per} A = k$ hold?

This problem and the corresponding problems about $\operatorname{per} A \leq k$ and $\operatorname{per} A \geq k$ are NP-hard, which is due to Valiant [Val79a]; we note that it is not known whether these problems actually belongs to NP.

Recall that determining the number of perfect matchings in a bipartite graph is equivalent to determining the permanent of an appropriate matrix, so that this problem is likewise NP-hard.

Restricted matching (cf. Section 14.7)

Let $G = (V, E)$ be a graph, and consider a decomposition of E into subsets E_i $(i = 1, \ldots, k)$. Also, let c and b_i $(i = 1, \ldots, k)$ be positive integers. Does there exist a matching K with c edges such that $|K \cap E_i| \leq b_i$ holds for $i = 1, \ldots, k$?

This problem is NP-complete, even when all b_i are 1; see [ItRT78].

Satisfiability (cf. Section 2.7)

Let $C_1 \ldots C_m$ be a formula involving n Boolean variables in conjunctive normal form. Does there exist an assignment of the values *true* and *false* to the n variables such that the given formula takes the value *true*?

This problem is NP-complete, even when each of the C_i involves precisely three of the n Boolean variables (3-SAT). This celebrated result due to [Coo71] was the starting point of the theory of NP-completeness.

Shortest cycle (cf. Sections 3.3 and 10.6)

Let $N = (G, w)$ be a network on a graph G, where w is a length function that may take negative values, and c an integer. Does G contain a cycle of length at most c?

This problem is NP-complete; see [GaJo79]. It can be solved in polynomial time for nonnegative length functions; see, for example, [ItRo78] and [Mon83]. Similar results hold for the analogous directed problem. Note that determining a cycle of minimum cycle mean is easy for arbitrary length functions w; see Section 10.6.

Shortest path (cf. Chapter 3 and Section 14.6)

Let s and t be two vertices in a network $N = (G, w)$ on a graph G, where w is a length function that may take negative values, and let c be an integer. Does there exist a path from s to t of length at most c?

This problem is NP-complete, and this also holds for the analogous directed problem; see [GaJo79]. As we saw in Section 14.6, the problem becomes polynomial if we assume that N does not contain any cycles of negative length. Particularly good algorithms exist for the special case where all edges have nonnegative length; see Chapter 3.

Spanning tree (cf. Chapter 4)

We know that a spanning tree in a connected graph G can be determined with linear complexity using either BFS or DFS; see Sections 3.3 and 8.2.

However, the problem usually becomes NP-complete if we add extra constraints such as either a lower bound on the number of leaves, or an upper bound on the maximal degree of the tree; see [GaJo79, §4.7]. The same conclusion holds if we ask whether the sum of the distances $d(u, v)$ in T (taken over all pairs (u, v) of vertices) can be bounded by c; see [JoLR78].

Steiner network (cf. Section 4.6)

Let $N = (G, w)$ be a network on a graph $G = (V, E)$, where $V = R \dot{\cup} S$ and where $w \colon E \to \mathbb{R}^+$ is a positive weight function, and let c be a positive integer. Does there exist a minimal spanning tree T for some induced subgraph whose vertex set has the form $R \cup S'$ with $S' \subset S$ so that $w(T) \leq c$?

This problem is NP-complete by a result of Karp [Kar72]. The problem becomes polynomial when either $|R|$ or $|S|$ is fixed.

Steiner tree (cf. Section 4.6)

For a given set of n points in the Euclidean plane, we want to find a minimal Steiner tree (that is, a tree of minimal length with respect to the Euclidean

distance) which contains the given n points. This problem was shown to be NP-hard by Garey, Graham and Johnson [GaGJ77].

Travelling salesman problem (TSP) (cf. Chapter 15)

Let $w : E \to \mathbb{R}^+$ be a positive length function on the complete graph K_n. Given a positive integer b, is there a tour (that is, a Hamiltonian cycle) of length at most b?

Recall that the TSP served as our standard example for an NP-complete problem. It remains NP-complete in the metric case, in the asymmetric case, and for length functions restricted to the values 1 and 2. The related questions of whether a tour is suboptimal or whether an optimal tour contains a given edge are likewise NP-hard.

The associated approximation problem is NP-hard in the general case, but easy in the metric case: there is a polynomial ε-approximative algorithm with $\varepsilon = 1/2$ for the ΔTSP. The existence of such an algorithm for a value $\varepsilon < 1/219$ would already imply P=NP; see [PaVe06].

See Chapter 15 and the monographs [LaLRS85] and [GuPa02].

Unextendable matching (cf. Section 7.2 and Chapter 13)

Let $G = (V, E)$ be an arbitrary graph, and let c be a positive integer. Does G contain an unextendable matching of cardinality at most c?

This problem is NP-complete by a result of Yannakakis and Gavril [YaGa80]. The problem remains NP-complete in the special cases of planar graphs and of bipartite graphs (even when the maximal degree is restricted to 3).

Recall that a matching which cannot be extended does not have to have maximal cardinality in general. As we have seen, it is easy to determine a *maximal* matching (that is, a matching of maximal cardinality) in G. Hence the existence of an unextendable matching of cardinality *at least* c is easy to decide.

Vertex cover (cf. Section 2.8)

Let $G = (V, E)$ be a graph, and let c be a positive integer. Does G have a vertex cover of cardinality at most c?

This problem is NP-complete by a result of Karp [Kar72]; see Theorem 2.8.2. It can be solved in polynomial time for perfect graphs (hence, for bipartite graphs); see [GrLS93].

Note that **Vertex cover** is equivalent to **Independent set**: the complement of a vertex cover is an independent set.

B

Solutions

People of quality know everything without ever having been taught anything.
MOLIÈRE

This appendix contains solutions (or extended hints) to virtually all the exercises. For difficult exercises, we include more details; if an exercise is of a purely computational nature, we usually state only the result.

B.1 Solutions for Chapter 1

1.1.2 As $2n-1$ is odd, $2i$ $(i = 1, \dots 2n-1)$ runs through all residue classes modulo $2n-1$. Therefore the sets F_i are pairwise disjoint. Clearly, each F_i is a factor of K_{2n}. As $F_1, \dots, F_{2n-1}$ contain altogether $n(2n-1)$ edges, they must form a factorization.

1.1.3 Note $T_3 = K_3$. The graph T_4 is K_6 with one 1-factor removed. The complement of T_5 is shown in Figure 1.12; cf. Exercise 1.5.11.

A vertex $\{x, y\}$ of T_n is adjacent precisely to the $2(n-2)$ vertices of the form $\{x, z\}$ and $\{y, z\}$, where $z \neq x, y$. Two distinct vertices $\{x, y\}$ and $\{x, z\}$ are adjacent precisely to the $n-3$ vertices $\{x, w\}$ for $w \neq x, y, z$ and to $\{y, z\}$. Finally, the common neighbors of two vertices $\{x, y\}$ and $\{z, w\}$, where x, y, z, w are distinct, are precisely $\{x, z\}$, $\{x, w\}$, $\{y, z\}$, and $\{y, w\}$.

1.1.4 For a given vertex x, there are exactly $a' = n - a - 1$ vertices which are not adjacent to x in G. If x and y are vertices adjacent in G, there are precisely $a - c - 1$ vertices which are adjacent to x but not to y, and precisely $(n - a - 1) - (a - c - 1)$ vertices which are adjacent neither to x nor to y. Thus $\overline{G}$ has parameters $a' = n - a - 1$ and $d' = n - 2a + c$. Similar arguments give $c' = n - 2a + d - 2$.

To prove the validity of the equation in question, choose some vertex x. Then there are $n - a - 1$ vertices z which are not adjacent to x. For each such vertex z, there are precisely d vertices y which are adjacent to x as well as to z. On the other hand, there are a vertices y adjacent to x, and for each such vertex y, there are $a - c - 1$ vertices z adjacent to y but not adjacent to x.

1.2.1 Let $W = (v_0, \ldots, v_n)$ be a walk with start vertex $a = v_0$ and end vertex $b = v_n$. If W is not a path, it contains repeated vertices. Let $x = v_i$ be the first such vertex, and let v_j be the next occurrence of x on W. Then the subwalk of W from v_i to v_j is a closed walk, and omitting it from W yields a shorter walk W' from a to b. Using induction on the length of W gives the assertion.

Now let $W = (v_0, \ldots, v_n)$ be a closed walk of odd length which is not a cycle. Suppose there exists some index $i \neq 0, n$ such that $v_0 = v_i = v_n$. Then one of the closed walks $(v_0, \ldots, v_i)$ or $(v_i, \ldots, v_n)$ has odd length, and the assertion follows by induction. In the general case, there are indices $i, j \neq 0, n$ with $i \neq j$ and $v_i = v_j$; again, the assertion follows by induction.

We obtain a closed walk of even length not containing any cycle if we append to some path P (from u to v, say) the same path P traversed in the opposite direction (that is, from v to u).

1.2.2 Let x and y be any two vertices. Then the connected components of x and y contain at least $(n+1)/2$ vertices each; hence they cannot be disjoint, and therefore they coincide.

1.2.3 Trivially, the condition is necessary. To show that it is also sufficient, choose some vertex s and let V_1 be its connected component. Then $V_2 = V \setminus V_1$ has to be empty: otherwise, the hypothesis would provide an edge vw with $v \in V_1$ and $w \in V_2$, and w would after all be in V_1, a contradiction.

1.2.4 If neither G nor $\overline{G}$ are connected, choose some vertex s and denote the connected components of G and $\overline{G}$ containing s by S and T, respectively. As each vertex $v \neq s$ is either adjacent to s in S or in T, we must have $V = S \cup T$. It can be seen by similar arguments that there cannot exist a pair (v, w) of vertices with $v \in S \setminus T$ and $w \in T \setminus S$, a contradiction.

1.2.5 The assertion follows from $\sum_v \deg v = 2n - 2$; see the proof of Lemma 1.1.1.

1.2.9 If $G \setminus e$ is connected, the assertion follows using induction on $|E|$. Otherwise, G consists of two connected components V_1 and V_2. Using induction on n, the assertion holds for the induced graphs $G|V_1$ and $G|V_2$. Hence

$$|E| \;=\; |(E|V_1)| + |(E|V_2)| + 1 \;\geq\; (|V_1| - 1) + (|V_2| - 1) + 1 \;=\; n - 1.$$

1.2.15 See Figure B.1.

1.2.16 The symbol u occurs precisely $\deg u - 1$ times in $\pi_V(G)$; this is similar to the proof of Lemma 1.2.12. In particular, stars are precisely those trees G for which all entries of $\pi_V(G)$ agree, whereas paths are the trees having a Prüfer code with distinct entries.

1.2.17 As a tree on n vertices has $n - 1$ edges, condition (1.6) is certainly necessary. By the solution to Exercise 1.2.16, the degree of a vertex u in a

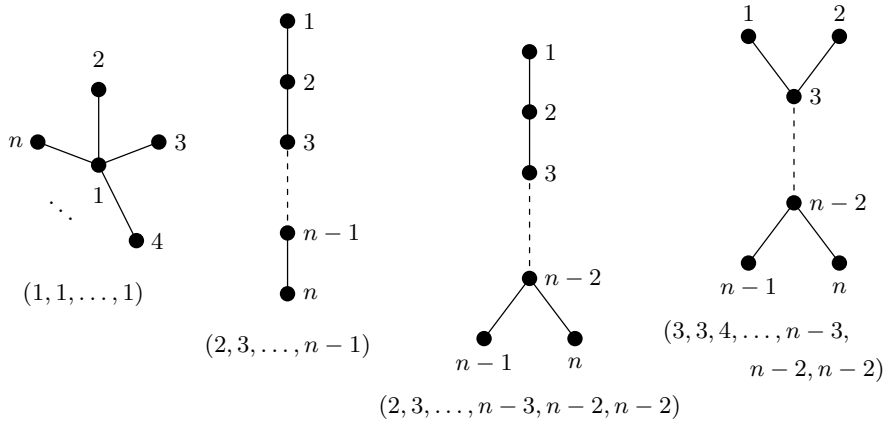


Fig. B.1. Solution to Exercise 1.2.15

tree T equals the number of entries u in the Prüfer code $\pi_V(T)$ plus 1. Now let $d_1, \ldots, d_n$ be a sequence of positive integers satisfying (1.6); then

$$(d_1 - 1) + (d_2 - 1) + \ldots + (d_n - 1) \;=\; n - 2.$$

Hence there are words of length $n-2$ over the alphabet $\{1, \ldots n\}$ which contain exactly $d_i - 1$ entries i (for $i = 1, \ldots n$), and the corresponding trees under the Prüfer code have the prescribed degree sequence. For the sequence $(1, 1, 1, 1, 2, 3, 3)$, we may use the Prüfer code $(5, 6, 6, 7, 7)$ to obtain the tree shown in Figure B.2.

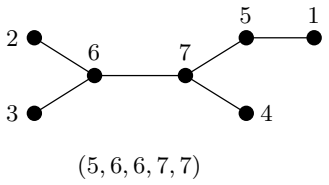


Fig. B.2. Tree with prescribed degree sequence

1.3.3 Denote the vertices of odd degree by x_i and y_i (for $i = 1, \ldots, k$). Adding the edges $x_i y_i$ to G yields an Eulerian multigraph H. The desired trails arise by omitting the edges $x_i y_i$ from a Euler tour for H.

1.3.4 Note that an edge uv of G has degree $\deg u + \deg v - 2$ when considered as a vertex of $L(G)$. In particular, $L(K_{m,n})$ is $(m+n-2)$-regular. If x, y, z, w are distinct vertices of $K_{m,n}$, edges of the form xy and zw are always adjacent to precisely two edges in $L(K_{m,n})$. Edges of the form xy and xz are adjacent

to $m-2$ or $n-2$ edges, depending on which part of $K_{m,n}$ contains x. Hence $L(K_{m,n})$ is an SRG if and only if $m = n$.

1.3.5 Note first that $L(G)$ is connected, since G is assumed to be connected. By Exercise 1.3.4, an edge uv of G has degree $\deg u + \deg v - 2$ in $L(G)$; by Theorem 1.3.1, $L(G)$ is Eulerian if and only if this number is always even. As G is connected, this requires that the degrees of all the vertices of G have the same parity. In particular, this condition is met if G is Eulerian: then all vertices of G have even degree. Finally, $L(K_{2n})$ is Eulerian while K_{2n} is not Eulerian, as all vertices have odd degree.

1.4.4 The existence of non-adjacent vertices u and v with $\deg u + \deg v < n$ would imply $m < \frac{1}{2}(n-2)(n-3)+n = \frac{1}{2}(n-1)(n-2)+2$, since the maximal number of edges arises if the remaining $n-2$ vertices induce a complete graph.

1.4.5 As K_6 is Hamiltonian, G also has to be Hamiltonian by Theorem 1.4.1. Therefore G contains a cycle of length 6. We have to add at least two edges to this cycle to obtain a graph where $\deg u + \deg v \geq 6$ holds for some pair of non-adjacent vertices u and v. On the other hand, it is easy to check that the closure of such a graph G is indeed K_6. Hence eight edges are needed.

1.4.6 Let $(e_1, \ldots, e_m)$ be an Euler tour of G; then the sequence $(e_1, \ldots, e_m, e_1)$ is a Hamiltonian cycle in $L(G)$. The converse is false; for example, K_4 is not Eulerian even though $L(K_4) = T_4$ is Hamiltonian.

1.4.8 We color the squares of a chess board alternately black and white, as usual. Note that a knight always moves from a black square to a white one, and from a white square to a black one; in the corresponding graph, all edges connect a black and a white vertex. (This means that G is bipartite; see Section 3.3.) Obviously, G can only contain a Hamiltonian cycle if the numbers of white and black vertices are equal, which is impossible if n and m are both odd. This accounts for case (a).

In case (b), the preceding necessary condition is satisfied. However, the cases $m = 1$ and $m = 2$ are trivially impossible. In order to show that a knight's cycle is also impossible for $m = 4$, we consider a second coloring of the chess board: the squares of the first and fourth rows are green, whereas the squares in rows two and three are red. Then a knight can move from a green square only to a red square; from a red square, green squares as well as red squares are accessible. Now assume the existence of a knight's cycle. Then the knight has to reach and to leave each of the green squares precisely once. As the green squares are only accessible from the red ones, the $2n$ moves from a red square always have to be moves to a green square, so that red and green squares alternate in the knight's cycle. But white and black squares also occur alternately; as the two colorings of the board are obviously distinct, this is impossible.

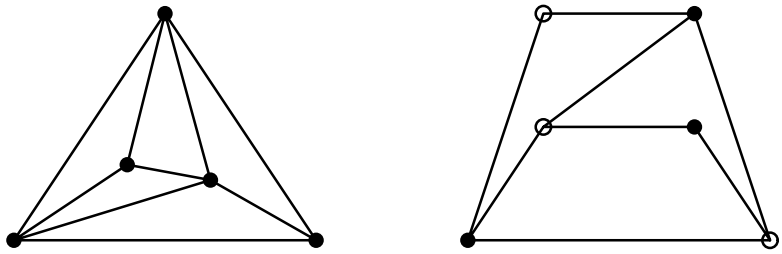


Fig. B.3. Plane realizations of $K_5 \setminus e$ and $K_{3,3} \setminus e$

1.5.6 See Figure B.3.

1.5.7 Any subdivision of a graph increases the number of vertices by the same value as the number of edges.

1.5.10 The Petersen graph G has girth $g = 5$. As G contains more than $40/3$ edges, G cannot be planar by Theorem 1.5.3. Figure B.4 shows G and a subgraph homeomorphic to $K_{3,3}$, where the vertices of $K_{3,3}$ are indicated by fat circles and squares, whereas the vertices obtained by subdivision are drawn as small circles. Thus Result 1.5.8 applies. Contracting each outer vertex of G with its adjacent inner vertex shows that G can be contracted to K_5, so that Result 1.5.9 likewise applies.

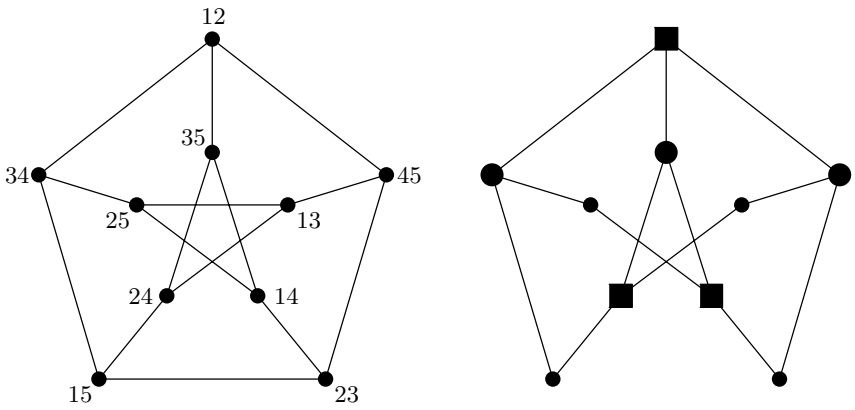


Fig. B.4. The Petersen graph

1.5.11 We write the 2-subsets $\{x, y\}$ of $\{1, \ldots, 5\}$ simply as xy. Then the vertices of T_5 are the xy, and xy and zw are adjacent in $\overline{T_5}$ if and only if x, y, z, w are four distinct elements. Now it is easy to give the desired isomorphism using the labelling of the vertices shown in Figure B.4.

1.5.12 Each permutation $\alpha \in S_5$ induces an automorphism of T_5 – and hence, by Exercise 1.5.11, an automorphism of the Petersen graph – by mapping each 2-subset xy to $x^\alpha y^\alpha$. Actually, S_5 already yields *all* automorphisms of the Petersen graph; however, proving this requires a little more effort. (Hint: Try to show that there are at most 120 automorphisms of the Petersen graph.)

1.5.13 For $n = 1, \ldots, 4$, K_n is already planar. For $n \geq 5$, K_n cannot be planar, since a planar graph on n vertices has at most $3n-6$ edges by Corollary 1.5.4. Thus we have to remove at least $\frac{1}{2}n(n-1)-(3n-6) = \frac{1}{2}(n-2)(n-5)+1$ edges. Using induction, it can be shown that there exists a planar graph with $3n-6$ edges for each n; in fact, this graph can even be assumed to have a triangle as its outer border. The induction basis ($n = 3, 4, 5$) and the recursive construction of placing a planar graph with n vertices and $3n-6$ edges inside a triangle are sketched in Figure B.5.

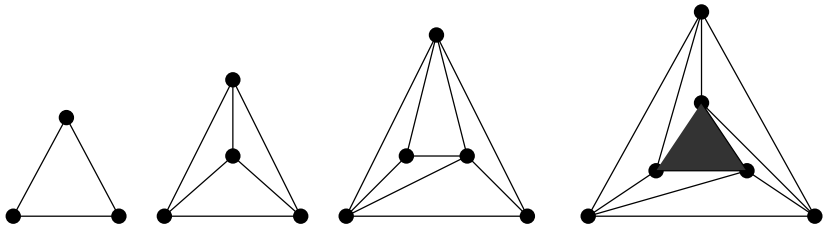


Fig. B.5. Maximal planar graphs

1.5.14 As $n - n_d$ vertices have degree at least $d+1$, Corollary 1.5.4 implies $(n-n_d)(d+1) \leq \sum_v \deg v = 2m \leq 6n-12$ and hence the assertion. In particular, $n_5 \geq 2$ and $n_6 \geq n/7$; thus more than 14% of the vertices of a planar graph have degree at most 6.

The given formula can be strengthened as follows: any planar graph can be embedded in a planar graph (by adding appropriate edges) whose vertices have degree at least 3. For these planar graphs, the left hand side of the inequality can be increased by $3n_d$, and we obtain

$$n_d \;\geq\; \frac{n(d-5)+12}{d-2};$$

in particular, $n_5 \geq 4$ and $n_6 \geq n/4$.

1.6.1 Let G be pseudosymmetric. Choose an arbitrary edge $e_1 = v_0v_1$, then some edge $e_2 = v_1v_2$ and so on, always selecting edges which have not occurred before. Whenever we reach a vertex $v_i \neq v_0$ via an edge e_i, there is an unused edge e_{i+1} available for leaving v_i, since G is pseudosymmetric. Hence our construction yields a directed cycle C. Removing C from G results in a pseudosymmetric graph H, and the assertion follows by induction.

1.6.4 Obviously, an edge contained in a cycle cannot be a bridge. Conversely, let $e = uv$ be an edge which is not a bridge. Then the connected component containing u and v is still connected after removing e, so that there exists a path P from u to v not containing e. Appending e to P yields the desired cycle.

1.6.5 G is Eulerian by Theorem 1.3.1. Let $(v_0, \dots, v_m = v_0)$ be the sequence of vertices in an Euler tour $(e_1, \dots, e_m)$ of G. Orienting each edge e_i from v_{i-1} towards v_i, we obtain an orientation of G, and $(e_1, \dots, e_m)$ is a directed Euler tour for this orientation. Hence this orientation is pseudosymmetric and strongly connected.

1.6.6 First let $W = (v_0, \dots, v_n)$ be a closed directed walk which is not a cycle. Suppose there exists some index $i \neq 0, n$ such that $v_0 = v_i = v_n$. Then each of the closed walks $(v_0, \dots, v_i)$ and $(v_i, \dots, v_n)$ has shorter length, and the assertion follows by induction. In the general case, there are indices $i, j \neq 0, n$ with $i \neq j$ and $v_i = v_j$; again, the assertion follows by induction.

Now let $W = (v_0, \dots, v_n)$ be a directed walk with start vertex $a = v_0$ and end vertex $b = v_n$. If W is not a path, it contains repeated vertices. Let $x = v_i$ be the first such vertex, and let v_j be the next occurrence of x on W. Then the subwalk W' of W from v_i to v_j is a directed closed walk and hence contains a directed cycle C. Omitting C from W yields a shorter directed walk W' from a to b, and the assertion follows by induction on the length of W.

B.2 Solutions for Chapter 2

2.1.3 Let G have the $n!$ permutations of $\{1, \dots, n\}$ as vertices, and let two permutations be adjacent if and only if they differ by only a transposition. The case $n = 3$ is shown in Figure B.6, where we denote the permutation (x, y, z) of $\{1, 2, 3\}$ by xyz; here the sequence $(123, 132, 312, 321, 231, 213)$ provides a solution.

2.1.4 (a) First assume that $G \setminus v_0$ is acyclic, and let C be a maximal path starting at v_0. Then C is a cycle. If C were not an Euler tour, we could find a cycle C' in $G \setminus C$ as in Example 2.1.2. By hypothesis, C' would have to contain v_0, so that C would not be maximal. Hence G is arbitrarily traceable from v_0. Conversely, suppose that G is arbitrarily traceable from v_0. If there exists a cycle C in $G \setminus v_0$, we can choose an Euler tour K of the connected component of v_0 in $G \setminus C$, so that K is a maximal trail starting in v_0, a contradiction.

(b) Let w be a vertex of maximal degree $2k$ in G, and let C be an Euler tour for G. Then C can be divided into k cycles $C_1, \dots, C_k$, each of which contains w only once. As $G \setminus v_0$ is acyclic by (a), v_0 has to occur in each of these cycles, and hence also $\deg v_0 = 2k$.

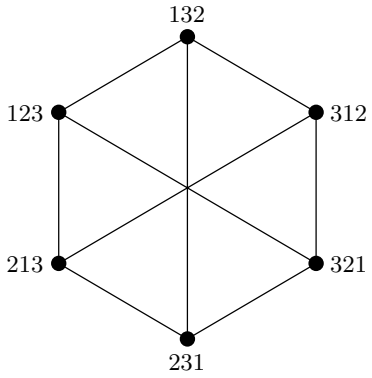


Fig. B.6. Transposition graph for S_3

(c) Suppose G is arbitrarily traceable from u, v, and w. By part (a), each of these three vertices has to occur in all cycles of G. Suppose that G contains at least two cycles (which intersect in u, v, and w); then it is easy to construct a third cycle which contains only two of these vertices, a contradiction. Hence G contains at most one cycle and thus is itself a cycle.

(d) By part (b), both vertices have to be vertices of maximal degree, say $2k$. Choose two vertices u and v and connect them by $2k$ parallel edges. Then all subdivisions of this multigraph are arbitrarily traceable from both u and v.

2.2.5 Use induction on h; the case $h = 1$ is clear. With $B = A^h$, the (i,k)-entry of A^{h+1} is the sum of all terms $b_{ij}a_{jk}$ over $j = 1, \ldots, n$. By the induction hypothesis, b_{ij} is the number of walks of length h from i to j. Moreover, $a_{jk} = 1$ if (j,k) is an edge, and $a_{jk} = 0$ otherwise. Observe that a walk of length $h+1$ from i to k consists of a walk of length h (from i to some vertex j) followed by a last edge (j,k). This proves the assertion for graphs; the same argument works in the directed case, if we restrict attention to directed walks.

2.2.6 By Exercise 2.2.5, the (i,j)-entry of the matrix A^2 is the number of walks of length 2 from i to j; note that this reduces to the degree of i whenever $i = j$. Denote the matrix with all entries equal to 1 by J. Using the defining properties of a strongly regular graph yields the desired quadratic equation: $A^2 = aI + cA + d(J - I - A)$.

2.3.2 Note that a word $w = a_i \ldots a_{i+n-1}$ is the immediate predecessor of a word $v = a_{i+1} \ldots a_{i+n}$ in a de Bruijn sequence if and only if the edge v has the end vertex of w as start vertex; thus the de Bruijn sequences correspond to Euler tours in $G_{s,n}$. It remains to show that $G_{s,n}$ satisfies the two conditions of Theorem 1.6.1. First, $G_{s,n}$ is strongly connected: two vertices $b_1 \ldots b_{n-1}$ and $c_1 \ldots c_{n-1}$ are connected by the directed path

$$(b_1 \dots b_{n-1}c_1,\ b_2 \dots b_{n-1}c_1c_2,\ \dots,\ b_{n-1}c_1 \dots c_{n-1}).$$

$G_{s,n}$ is also pseudosymmetric: $d_{\text{in}}(x) = d_{\text{out}}(x) = s$ for each vertex x.

2.3.3 The digraph $G_{3,3}$ is shown in Figure B.7.

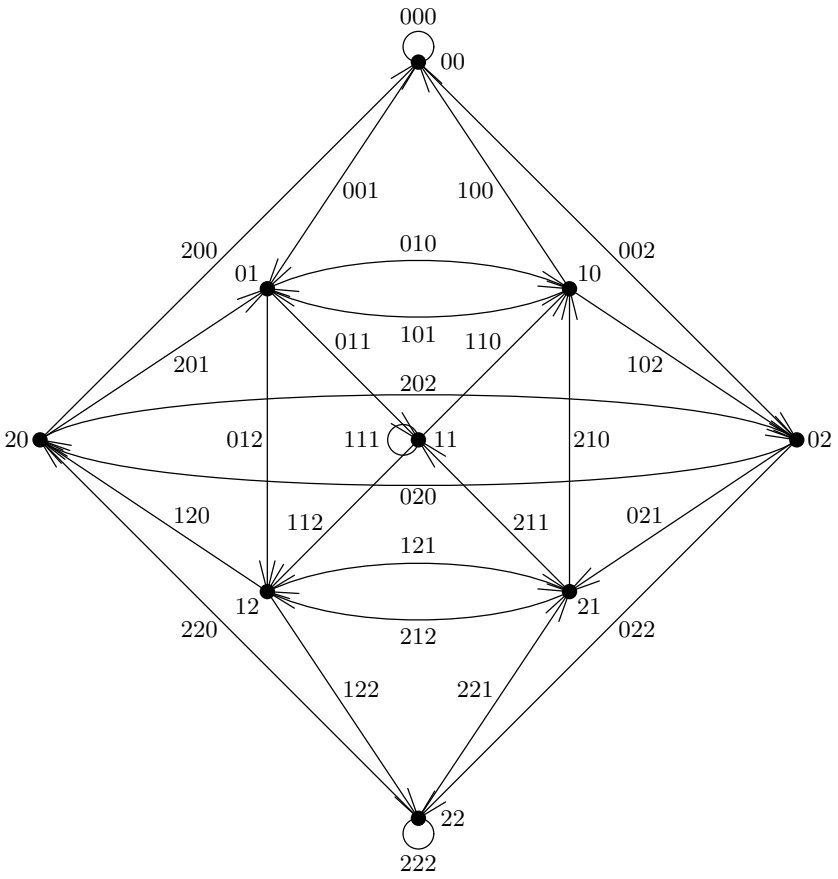


Fig. B.7. The digraph $G_{3,3}$

Using $s = 00$, the procedure TRACE$(s, \text{new}; K)$ yields the cycle

$$K \ = \ (000, 001, 010, 100, 002, 020, 200).$$

Then all edges with start vertex 00 have been used, and $L = (00, 01, 10, 02, 20)$. In step (5) of EULER, the vertex $u = 20$ is removed from L; then step (7) calls TRACE$(u, \text{new}; C)$, which yields the cycle

$$C \ = \ (201, 011, 110, 101, 012, 120, 202, 021, 210, 102, 022, 220).$$

This cycle is inserted in front of the edge 200 into K according to step (8) of EULER; we then have $K = (000, 001, \ldots, 020, 201, 011, \ldots, 220, 200)$ and $L = (00, 01, 10, 02, 11, 12, 21, 22)$. Next, $u = 22$ is removed from L in step (5), and the cycle

$$C = (221, 211, 111, 112, 121, 212, 122, 222)$$

is constructed and inserted into K in front of the edge 220. After this, EULER discovers that all edges have been used (by investigating all vertices in L). The de Bruijn sequence corresponding to this Euler tour is

$$0\,0\,0\,1\,0\,0\,2\,0\,1\,1\,0\,1\,2\,0\,2\,1\,0\,2\,2\,1\,1\,1\,2\,1\,2\,2\,2.$$

2.6.8 Let $G = (V, E)$ be the empty digraph with n vertices: $E = \emptyset$. Then any algorithm using the adjacency matrix has to check at least one of the two entries a_{ij} and a_{ji} for each pair (i, j) with $i \neq j$: otherwise, we could add the edges (i, j) and (j, i) to G and the algorithm would not realize that the digraph is no longer acyclic. Thus the algorithm has to check at least $n(n-1)/2 = \Omega(n^2)$ entries.

2.6.9 The algorithm first calculates $\mathrm{ind}(1) = 2$, $\mathrm{ind}(2) = 0$, $\mathrm{ind}(3) = 3$, $\mathrm{ind}(4) = 1$, $\mathrm{ind}(5) = 2$, $\mathrm{ind}(6) = 4$, $\mathrm{ind}(7) = 3$, and $L = (2)$. Then 2 is removed from L and the function *ind* is updated as follows: $\mathrm{ind}(1) = 1$, $\mathrm{ind}(3) = 2$, $\mathrm{ind}(4) = 0$, $\mathrm{ind}(7) = 2$. Now 4 is appended to L. During the next iteration, 4 is removed from L, and the following updates are performed: $\mathrm{ind}(1) = 0$, $\mathrm{ind}(3) = 1$, $\mathrm{ind}(5) = 1$, $\mathrm{ind}(7) = 1$. Then 1 is appended to L and immediately removed again during the next iteration. Continuing in this way yields the topological sorting $(2, 4, 1, 3, 5, 7, 6)$ for G; see Figure B.8, where indeed all edges are oriented from left to right.

2.7.6 DHC contains HC as a special case; this follows by considering the complete orientation of a given graph.

2.7.7 We transform HC to HP. Let $G = (V, E)$ be a connected graph. We choose a fixed vertex v_0. Then we adjoin three new vertices u, u', and w to G, and add the following edges: uu', wv_0, and an edge uv for each vertex v adjacent to v_0; see Figure B.9. The resulting graph G' has a Hamiltonian path if and only if G admits a Hamiltonian cycle; this follows by noting that every Hamiltonian path of G' has to start with the edge uu' and to end with the edge v_0w.

2.7.8 Note that HP is a special case of **Longest path**: given a graph G with n vertices, we apply **Longest path** with $k = n$.

Now assume that we also have to specify the end vertices of the path. If we had a polynomial algorithm for this modified problem, we could just invoke

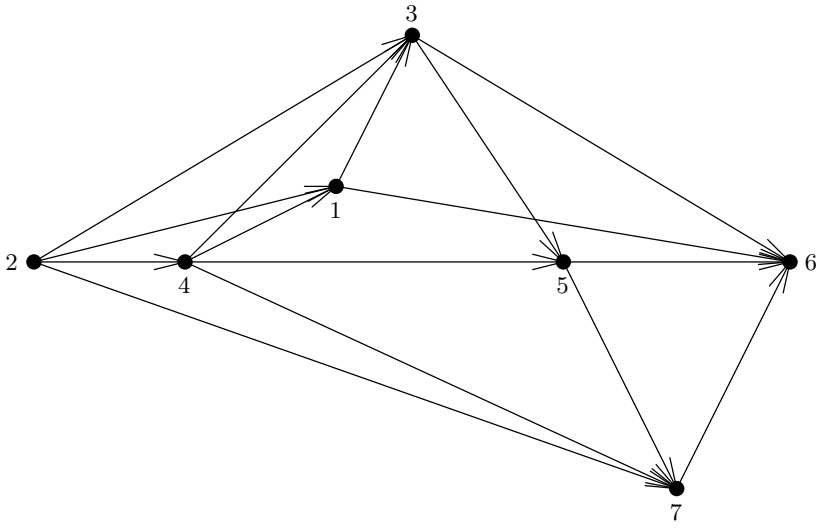


Fig. B.8. Solution to Exercise 2.6.9

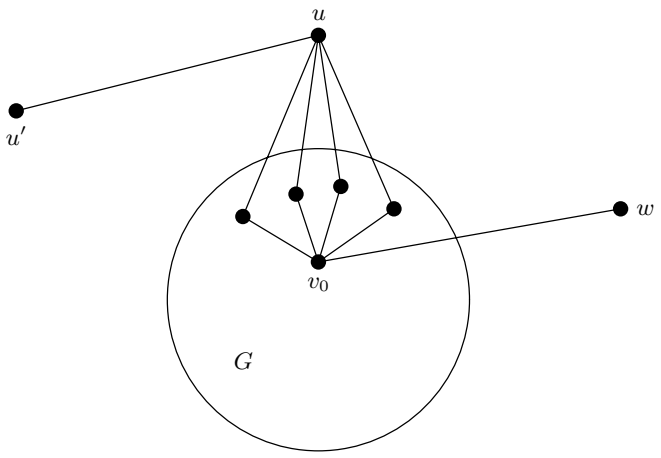


Fig. B.9. Construction for Exercise 2.7.7

the algorithm for all pairs of vertices to get a polynomial algorithm for the unrestricted problem.

The corresponding result holds for longest cycles: the question "Does a given graph G admit a cycle consisting of at least k edges?" contains HC.

2.8.3 Independent sets are precisely the complements of vertex covers. As VC is NP-complete, it follows immediately that IS is NP-complete as well.

The cliques in a graph G are precisely the independent sets of the complementary graph $\overline{G}$. Therefore, **Clique** is likewise NP-complete.

B.3 Solutions for Chapter 3

3.1.3 Let all pairs (j,k) with $j = 1,\ldots,n$ and $k = 0,\ldots,b$ be vertices of G. We choose all pairs $((j-1,k),(j,k))$ as edges of length 0 (for $j = 2,\ldots,n$; $k = 0,\ldots,b$), and all pairs $((j-1,k_j-a_j),(j,k_j))$ as edges with length c_j (for $j = 2,\ldots,n$ and $k_j = a_j,\ldots,b$). We also adjoin a start vertex s to G, and add the edges $(s,(1,0))$ with length 0 and $(s,(1,a_1))$ with length c_1. Then the paths from s to (j,k) correspond to those subsets of $\{1,\ldots,j\}$ whose total weight is k (and whose total value is the length of the associated path). Finally, we add an end vertex t and edges $((n,k),t)$ of length 0 (for $k = 0,\ldots,b$). Then paths from s to t correspond to subsets whose weight is at most b, and the length of a longest path from s to t is the value of an optimal solution for the given knapsack problem.

3.2.3 The distances in the metric space have to be integral; moreover, $d(x,y) \geq 2$ always has to imply that a point z with $d(x,y) = d(x,z) + d(z,y)$ exists. It is clear that this condition is necessary. In order to show that it is also sufficient, choose all pairs $\{x,y\}$ with $d(x,y) = 1$ as edges.

3.3.4 The connected components can be determined as follows, where p denotes the number of connected components and where $c(v)$ is the component of G containing $v \in V$.

Procedure COMP$(G;p,c)$

(1) $i \leftarrow 1$;
(2) **while** $V \neq \emptyset$ **do**
(3) choose a vertex $s \in V$
(4) BFS$(G,s;d)$
(5) $L \leftarrow \{v \in V : d(v)$ is defined$\}$; $V \leftarrow V \setminus L$;
(6) **for** $v \in L$ **do** $c(v) \leftarrow i$ **od**
(7) $i \leftarrow i+1$
(8) **od**

3.3.8 Let G be a graph containing cycles. Obviously, G contains a cycle which is accessible from some vertex s if and only if a BFS with start vertex s reaches a vertex w (when searching from the vertex v, say) such that $d(w)$ is already defined. Considering the point where such a vertex w occurs for the first time, we obtain a bound g for the length of a shortest cycle accessible from s:[1]

$$g \leq \begin{cases} 2d(v)+2 & \text{if } d(w) = d(v)+1; \\ 2d(v)+1 & \text{if } d(w) = d(v). \end{cases}$$

If $d(w) = d(v)$, the bound cannot be improved by continuing the BFS. However, if $d(w) = d(v)+1$, the BFS should be continued until all vertices which are in the same layer as v have been examined, because l might still be decreased by one; after this, the BFS may be terminated.

If we execute this procedure for all possible start vertices, the final value of g clearly equals the girth of G. If we also store a vertex s for which the BFS did yield the best value for g, it is easy to actually determine a cycle C of shortest length using a final modified BFS with start vertex s: we always store the vertex v from which w is reached when it is labelled with $d(w)$; that is, we add the instruction $p(w) \leftarrow v$ in step (7) of BFS. The final BFS can be terminated as soon as an edge vw which closes a cycle C occurs. Then we use the predecessor function p to construct the paths (in the BFS-tree T_s) from v and w to the root, and define C as the union of these two paths and the edge vw. We leave it to the reader to write down such a procedure explicitly.[2] As BFS has complexity $O(|E|)$, we achieve a complexity of $O(|V||E|)$ by this approach.

3.4.5 As the distances $d(s,v)$ are known by assumption, one may determine with complexity $O(|E|)$ the set E'' of *all* edges of G satisfying condition (3.2). It follows from the proof of Theorem 3.4.4 that E'' contains an SP-tree; more precisely, any spanning arborescence with root s of $G'' = (V, E'')$ is an SP-tree. Hence a BFS on G'' with start vertex s will determine the desired SP-tree. In view of Theorem 3.3.2, this proves the assertion.

3.4.6 First let T be an SP-tree and uv an edge of G. By definition, the path from s to v in T is a shortest path from s to v in G. On the other hand, appending the edge uv to the path from s to u in T also yields a path from s to v in G. Therefore

$$d_T(s,v) \;=\; d(s,v) \;\leq\; d_T(s,u) + w(uv),$$

[1] Note that this is indeed just a bound; the precise length can be determined by backtracking the paths from v and w to s in the BFS-tree T_s up to the first vertex they have in common. Obviously, this vertex does not have to be s.

[2] If we want to check first whether G actually contains cycles, we may use the procedure COMP of Exercise 3.3.4 to determine the connected components, and then check the numbers of edges of the components using Theorem 1.2.8.

which is the desired inequality. Conversely, suppose that

$$(*) \qquad d_T(s,v) \;\leq\; d_T(s,u) + w(uv)$$

holds for all edges uv of G. If P is a shortest path from s to v in G (for $v \neq s$) and $e = uv$ is the last edge of P, then $P' = P \setminus e$ is a shortest path from s to u in G, by Lemma 3.4.1. Using induction on the number of edges of P, we may assume $d(s,u) = w(P') = d_T(s,u)$. Then $(*)$ implies

$$d(s,v) \;=\; d(s,u) + w(uv) \;=\; d_T(s,u) + w(uv) \;\geq\; d_T(s,v),$$

so that $d_T(s,v) = d(s,v)$. Thus T is indeed a shortest path tree.

3.4.7 Consider the network (G,w) displayed in Figure B.10. Then

$$P \;=\; s - a - b - c$$

is the unique shortest path from s to c (with length 3), and $P' = s - c - a$ is the unique shortest path from s to a (with length 0). Hence any SP-tree would have to contain the union of these two paths; but this union is already all of G and contains, for instance, the directed cycle $C = a - b - c - a$, a contradiction.

Now change the value $w(ca)$ from -4 to -3, so that C is still a directed cycle of negative length. But now $d(s,a) = 1$, and both P' and $s - a$ are shortest paths from s to a. Thus the path P is an SP-tree for the modified network.

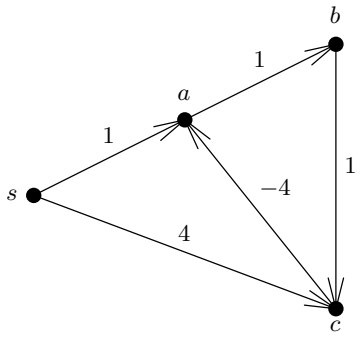


Fig. B.10. A network with a negative cycle

3.5.2 Let us again consider the network (G,w) used for the solution of Exercise 3.4.7; see Figure B.10. This time, we change the value $w(ca)$ from -4 to -2, so that $C = a - b - c - a$ becomes a directed cycle of length 0. Then $d(s,s) = 0$, $d(s,a) = 1$, $d(s,b) = 2$ and $d(s,c) = 3$ in the modified network,

giving one solution of Bellman's equations (B). However, it is easily checked that $u_s = u_a = 0$, $u_b = 1$ and $u_c = 2$ also gives a solution of (B).

It is easy to generalize this example: Let (G, w) be any network containing an induced cycle C of length 0, and assume that no edges are leading from C to another vertex of G. By subtracting a suitable constant from the distances of all vertices on C, one may obtain a second solution of system (B).

3.5.5 Let u_i denote the length of a longest path from 1 to i. Then the following analogue of the Bellman equations has to be satisfied:

$$(\mathrm{B}') \qquad u_1 = 0 \quad \text{and} \quad u_i = \max\{u_k + w_{ki} : i \neq k\} \qquad (i = 2, \ldots, n),$$

where we put $w_{ki} = -\infty$ if (k, i) is not an edge of G. Then the results of Section 3.5 carry over to this case: replace w by $-w$ and apply the original theorems to $(G, -w)$. If we do not want to require G to be acyclic, it suffices to assume that G contains cycles of negative length only.

The digraph corresponding to the knapsack problem of Exercise 3.1.3 is acyclic, so that it is possible to determine a longest path from s to t – that is, a solution of the knapsack problem – with complexity $O(|E|)$. However, this does not yield an efficient algorithm, because the number of edges of G has order of magnitude $O(nb)$, so that it depends not only on n but also on b. Restricting the values of b yields a polynomial algorithm, whereas the general knapsack problem is NP-hard; see [Kar72] and [GaJo79].

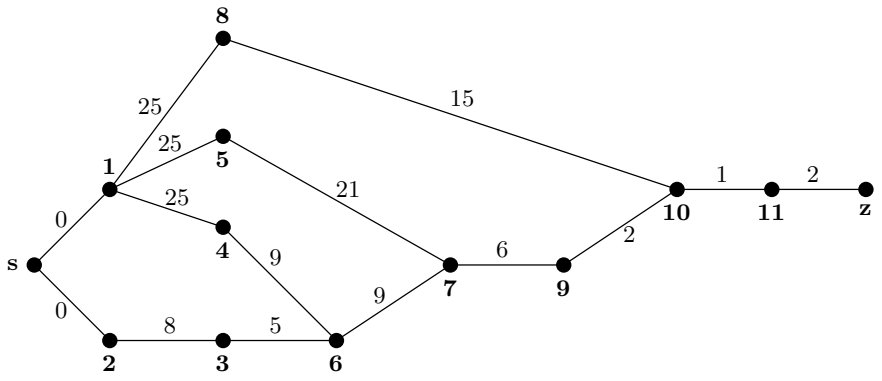


Fig. B.11. Digraph for the project *New production facility*

3.6.2 We obtain the network shown in Figure B.11 and the values $t_s = 0$, $t_1 = 0$, $t_2 = 0$, $t_3 = 8$, $t_4 = 25$, $t_5 = 25$, $t_8 = 25$, $t_6 = 34$, $t_7 = 46$, $t_9 = 52$, $t_{10} = 54$, $t_{11} = 55$, $t_z = 57$ and $T_z = 57$, $m_z = 0$; $T_{11} = 55$, $m_{11} = 0$; $T_{10} = 54$, $m_{10} = 0$; $T_9 = 52$, $m_9 = 0$; $T_7 = 46$, $m_7 = 0$; $T_6 = 37$, $m_6 = 3$; $T_8 = 39$, $m_8 = 14$; $T_5 = 25$, $m_5 = 0$; $T_4 = 28$, $m_4 = 3$; $T_3 = 32$, $m_3 = 24$;

$T_2 = 24$, $m_2 = 24$; $T_1 = 0$, $m_1 = 0$; $T_s = 0$, $m_s = 0$. The critical path is $(s, 1, 5, 7, 9, 10, 11, z)$.

3.6.3 Consider the network on G where all edges have length 1. As G is acyclic, we may use TOPSORT to determine a topological sorting for G. Then the length of a longest path from s to v can be determined as in Section 3.6 or as explained in the solution to Exercise 3.5.5, by recursively solving the equations (B′) or (CPM), respectively. The whole method has complexity $O(|E|)$.

3.6.4 For the time being, we denote the rank function on G by r'. Thus we have to show that, at the end of RANK, $r(v) = r'(v)$ holds for all v. This can be done using induction on the order in which r is defined. Note that $p(w)$ is the predecessor of w on a longest path from s to w; this function can also be used to find such a path: in reverse order, we get the path $(w, p(w), p(p(w)), \ldots, s)$. The values $d(v) = d_{\text{in}}(v)$ needed in step (3) can be determined from the adjacency lists (as in TOPSORT). Ordering the vertices of G by increasing rank yields a topological sorting of G; the order of vertices of the same rank is arbitrary. As each edge is examined exactly twice during RANK (once when d is determined in (3), and once in step (7)), this algorithm has complexity $O(|E|)$.

3.7.3 We introduce a variable $p(v)$ which will yield the predecessor of v on a shortest path from s to v (at the end of the algorithm): $p(v)$ is initialized to be 0, and step (6) is changed as follows:

(6′) **for** $v \in T \cap A_u$ **do if** $d(u) + w_{uv} < d(v)$
then $d(v) \leftarrow d(u) + w_{uv}$; $p(v) \leftarrow u$ **fi od**

3.7.5 One obtains in turn $d(1) = 0$, $d(5) = 1$, $d(3) = 2$, $d(4) = 6$, $d(2) = 9$, $d(8) = 13$, $d(6) = d(7) = 14$.

3.7.8 We may assume the given network to be connected; then planarity implies $|E| = \Theta(|V|)$; see Example 2.5.1. Thus the modified algorithm of Dijkstra has complexity $O(|V| \log |V|)$.

3.7.9 Let us denote the values defined in (1) and (2) by $d_0(v)$, and the values defined during the k-th iteration of the **repeat**-loop by $d_k(v)$. Using induction, one shows that $d_k(v)$ is the length of a shortest path from s to v which has at most k edges. As (G, w) does not contain any cycles of negative length, a shortest path from s to v consists of at most $|V| - 1$ edges. Thus the condition in (7) holds for $k = |V|$ at the latest. As one iteration of the **repeat**-loop requires $O(|E|)$ steps (using backward adjacency lists), we obtain a complexity of $O(|V||E|)$.

3.8.1 Determine the least common multiple T of all time cycles and replace each line L with time cycle $T_L = T/m_L$, where $m_L \neq 1$, by m_L lines with time cycle T and times of departure s_L, $s_L + T_L$, $s_L + 2T_L$,...

3.9.3 Proceed as in the solution to Exercise 3.7.3.

3.9.5 The final matrix is

$$D_7 = \begin{pmatrix} 0 & 4 & 5 & 7 & 12 & 10 & 12 \\ \infty & 0 & 6 & 3 & 8 & 6 & 8 \\ \infty & \infty & 0 & 4 & 9 & 7 & 9 \\ \infty & \infty & 3 & 0 & 5 & 3 & 3 \\ \infty & \infty & 7 & 4 & 0 & 3 & 2 \\ \infty & \infty & 9 & 6 & 2 & 0 & 2 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 \end{pmatrix}.$$

3.9.6 Replace the length function w in the procedure FLOYD by the *adjacency function* of G: put $d(i,j) = 1$ in (3) if ij is an edge, and $d(i,j) = 0$ otherwise. Then change step (9) to

(9′) $\quad d(i,j) \leftarrow \max\left(d(i,j), \min\left(d(i,k), d(k,j)\right)\right);$

alternatively, *max* could be interpreted as the Boolean operation *or*, and *min* as *and*.

3.9.7 Let G be an acyclic digraph, and consider the network on G having all lengths equal to 1. As in the solution to Exercise 3.9.6, we replace the length function w in the procedure FLOYD by the adjacency function of G; moreover, we calculate $\max\left(d(i,j), d(i,k) + d(k,j)\right)$ in step (9) of that procedure instead of the minimum given there. As G is acyclic, the revised procedure will compute longest paths between all pairs of vertices: at the end of the algorithm, $d(i,j) = 0$ if and only if j is not accessible from i; otherwise, $d(i,j)$ is the maximal length of a path from i to j. (This can be shown by analogy to the proof of Theorem 3.9.2.) Then G_{red} consists of all the edges ij with $d(i,j) = 1$.

3.10.3 Define the values of the variables $d_k(v)$ as in the solution to Exercise 3.7.9. Then G contains a directed cycle of negative length which is accessible from s if and only if $d_{n-1} \neq d_n$. (The reader should prove this claim in detail.) Since s is a root of G, the algorithm of Bellman-Ford can be used to find cycles of negative length by replacing the **repeat-until**-loop used in BELLFORD with a **for-do**-loop. If we also introduce a predecessor function $p(v)$, we can find either a directed cycle of negative length or an SP-tree with root s. We give such a procedure below, using backward adjacency lists A'_v.

Procedure SPTREE$(G, w, s; d, p, \mathrm{neg}, T)$

```
(1)  d(s) ← 0;
(2)  T ← ∅;
(3)  for v ∈ V \ {s} do d(v) ← ∞ od
(4)  for i = 1 to n do
(5)       for v ∈ V do d'(v) ← d(v) od
(6)       for v ∈ V do
(7)            for u ∈ A'_v do
(8)                 if d'(v) > d'(u) + w_uv
(9)                 then d(v) ← d'(u) + w_uv, p(v) ← u
(10)                fi
(11)           od
(12)      od
(13) od
(14) if d(v) = d'(v) for all v ∈ V
(15) then neg ← false;
(16)      for v ∈ V \ {s} do T ← T ∪ {p(v)v} od
(17) else neg ← true
(18) fi
```

3.10.4 Replace the initial values $d(i,i) = 0$ in step (3) of procedure FLOYD by $d(i,i) = \infty$. Then, at the end of the algorithm, $d(i,i)$ equals the shortest length of a directed cycle through i.

3.11.2 Note that $a = a \oplus o$ shows that $\succeq$ is reflexive. Also $a = b \oplus b'$ and $b = c \oplus c'$ imply $a = c \oplus (b' + c')$, so that $\succeq$ is transitive as well. Suppose that $\oplus$ is idempotent. Then $a = b \oplus c$ and $b = a \oplus d$ imply

$$a = b \oplus c = b \oplus (b \oplus c) = b \oplus a = a \oplus b = a \oplus (a \oplus d) = a \oplus d = b;$$

it follows that $\succeq$ is antisymmetric.

3.11.3 Let E be the matrix with diagonal entries 0 and all other entries ∞. Then $D = D * W \oplus E$.

3.11.5 We have $(A')^k = \sum_{i=0}^{k} \binom{k}{i} A^i = \sum_{i=0}^{k} A^i = A^{(k)}$. Thus $A^{(n)}$ can be calculated for $n = 2^a$ using a matrix multiplications:

$$A^{(1)} = A' = A \oplus E,\ A^{(2)} = (A')^2,\ A^{(4)} = (A^{(2)})^2,\ \text{etc.}$$

If we assume that the operations $\oplus$ and $*$ in R take one step each, we obtain a complexity of $O(n^3 \log n)$ for this method of calculating $A^{(n)}$. For the special case $(\overline{\mathbb{R}}, \oplus, *)$, we get – as explained in Lemma 3.11.4 – an alternative to the algorithm of Floyd-Warshall, as $D = W^{(n-1)}$. However, the complexity of this technique is inferior to the one achieved in Theorem 3.9.2.

3.11.6 It is routine to verify that the matrices form a path algebra. For any solution Y of Equation (3.8), we have

$$Y \;=\; W * (W * Y \oplus B) \oplus B \;=\; W^2 * Y \oplus W^{(1)} * B;$$

hence, by induction,

$$Y \;=\; W^{k+1} * Y \oplus W^{(k)} * B \qquad \text{for all } k.$$

In particular, for $k = p$,

$$Y \;=\; W^{p+1} * Y \oplus W^* * B; \qquad \text{that is, } \; Y \succeq W^* * B.$$

If the addition $\oplus$ on R is idempotent, then addition of matrices is likewise idempotent; in this case, the corresponding preordering on the set of matrices is even a partial ordering by Exercise 3.11.2. Then the minimal solution $W^* * B$ of $(*)$ is unique.

3.11.10 Choose $R = \{a : 0 \leq a \leq 1\}$, $\oplus = \max$, and $* = \cdot$.

3.11.11 Note that A is stable if and only if $A^r = 0$ for some $r \in \mathbb{N}$, since $A^{(r-1)} = A^{(r)} = A^{(r-1)} + A^r$ holds if and only if $A^r = 0$. Lemma 3.11.4 implies that this condition is satisfied in the acyclic case: then each walk contains at most $r - 1$ edges, where r is the number of vertices of G. In this case, A is a solution of the equation $A^* = A^*A + E$. As K is a field, this means $A^*(E - A) = E$; that is, $A^* = (E - A)^{-1}$.

More generally, it is possible to show that A is stable if all cycles in G have weight 0 with respect to w. The converse is false in general: it is easy to find an example with weights 1 and -1 such that A is stable, but G contains cycles of weight $\neq 0$. However, the converse does hold for $K = \mathbb{R}$ and positive weights.

B.4 Solutions for Chapter 4

4.1.2

(1) ⇒ (2): Let e be an edge contained in the unique cycle C of G. Then $G \setminus e$ is connected and acyclic, so that $G \setminus e$ is a tree.

(2) ⇒ (3): As every tree on n vertices has $n - 1$ edges and is connected, the claim in (3) follows.

(3) ⇒ (4): As G is not a tree (since it has one more edge than a tree would have), there must be edges in G which are not bridges; see Lemma 4.1.1. Removing some edge e which is not a bridge yields a tree, so that e has to be contained in each cycle of G. Thus the set of all edges which are not bridges forms a cycle.

(4) $\Rightarrow$ (1): An edge e is not a bridge if and only if it lies in a cycle; see Exercise 1.6.4). Thus G contains a unique cycle, which consists of those edges which are not bridges.

4.1.3 The claim concerning the number of centers is clear for the trees K_1 and K_2. For every other tree T, remove all leaves of T; then the resulting tree T' has the same centers as T, and the assertion follows by induction.

Denote the diameter of a tree T by d and the eccentricity of a center by e. Then either $d = 2e$ or $d = 2e - 1$, and $d = 2e$ holds if and only if T has a unique center. For a formal proof, proceed again by induction.

4.1.4 Let W be a trail of maximal length in G. As G is acyclic, W has to be a path, and as W is maximal, the end vertices of W have degree 1. Thus $G \setminus W$ is a forest containing $2k-2$ vertices of odd degree. Now use induction.

4.1.5 By hypothesis, $\overline{T}$ has at least two connected components. Let x and y be two arbitrary vertices in distinct connected components of $\overline{T}$. In particular, x and y are not adjacent in $\overline{T}$, so that T contains the edge xy. Thus any two points in distinct components of $\overline{T}$ have to be adjacent in T.

The preceding observation shows that there cannot be three distinct connected components: otherwise, we would obtain a cycle of length 3 in T. Moreover, one of the two components must be an isolated point of $\overline{T}$: otherwise, we would obtain a cycle of length 4 in T. Hence T contains a vertex x which is adjacent to all other vertices, so that T is a star. The final assertion follows from Exercise 1.2.4 and Theorem 1.2.6.

4.1.6 There are precisely six isomorphism types of trees on 6 vertices; representatives for these types were given in Figure 1.6; we will denote these representatives by $T_1, \ldots, T_6$. Now let T be any tree on $\{1, \ldots, 6\}$. Then the image of T under an arbitrary permutation $\sigma \in S_6$ is a tree isomorphic to T. By a well-known equation for permutation groups, the number of trees isomorphic to T is equal to the order of S_6 (that is, $6! = 720$) divided by the order of the automorphism group of T. We obtain:

T_1: cyclic group of order 2 (rotate the tree by 180°), 360 isomorphic trees;
T_2: cyclic group of order 2 (exchange the two lower leaves of the tree), 360 isomorphic trees;
T_3: symmetric group S_3 (acting on the three lower leaves of the tree), 120 isomorphic trees;
T_4: cyclic group of order 2 (reflect the tree, exchanging the two branches), 360 isomorphic trees;
T_5: direct product of 3 cyclic groups of order 2 (reflect the tree, exchanging the two centers and the two pairs of leaves; or switch the two leaves of one of the two pairs), 90 isomorphic trees;
T_6: symmetric group S_5 (acting on the five leaves), 6 isomorphic trees.

This gives a total of $360 + 360 + 120 + 360 + 90 + 6 = 1296 = 6^4$ trees, which agrees with the result of Corollary 1.2.11.

4.2.11 By Theorem 4.2.9, the number of spanning trees of the complete bipartite graph $K_{m,n}$ is equal to the absolute value of the determinant of the matrix

$$A' = \begin{pmatrix} nI_m & -J_{m,n-1} \\ -J_{n-1,m} & mI_{n-1} \end{pmatrix},$$

where the indices give the numbers of rows and columns of the respective submatrices (and where I denotes an identity matrix and J a matrix having all entries 1, as usual). Now it is just a matter of some linear algebra to show $\det A' = n^{m-1}m^{n-1}$: using appropriate row and column transformations, one can transform A' into a triangular matrix with diagonal entries $1, n \ldots, n, m, \ldots, m$.

4.2.12 The proofs of the results in question carry over: just take into account that now $1 + 1 = 0$, and hence $-1 = +1$.

4.2.13 First assume that G' is bipartite, with respect to the partition $V = S \dot\cup T$. Let M' be a square submatrix of M of order k, say. The case $k = 1$ is trivial, so let $k \neq 1$. First consider the case where each column of M' contains two entries 1. The k vertices corresponding to the rows of M' can be divided into two sets $S' \subset S$ and $T' \subset T$. Each column of M' corresponds to an edge of G which has both end vertices in $S' \cup T'$ (by hypothesis). As G is bipartite, each column of M has one entry 1 in a row corresponding to S', and the other entry 1 in a row corresponding to T'. Hence the sum of the rows corresponding to S' equals the sum of the rows corresponding to T', so that the rows of M' are linearly dependent, and hence $\det M' = 0$. It remains to consider the case where M' contains a column with at most one entry 1. Then the claim follows by developing $\det M'$ with respect to this column (and using induction).

Conversely, let M be totally unimodular, and suppose that G is not bipartite. By Theorem 3.3.5, G contains a cycle C of odd length, say

$$C: \quad v_0 \quad \overline{\quad e_1 \quad} \quad v_1 \quad \ldots \quad v_{2n-1} \quad \overline{\quad e_{2n} \quad} \quad v_{2n} \quad \overline{\quad e_{2n+1} \quad} \quad v_0.$$

But then the determinant of the submatrix M corresponding to the $2n + 1$ vertices and the $2n + 1$ edges of C is 2, a contradiction.

4.2.14 By Corollary 1.2.11, the graph K_n has precisely n^{n-2} spanning trees. Note that each spanning tree of K_n has $n - 1$ edges and that each edge e has to be contained in the same number x of spanning trees, which implies $x = 2n^{n-3}$. Hence the number of spanning trees of $K_n \setminus e$ is $n^{n-2} - 2n^{n-3} = (n - 2)n^{n-3}$.

4.2.15 G has $p = n - m$ connected components.

4.3.4 Let e be an edge incident with v which has smallest weight among all such edges, and suppose that e is not contained in a given minimal spanning tree T for G. The cycle $C_T(e)$ which arises by adding e to T has to contain a second edge incident with v, say f; by Theorem 4.3.1, $w(e) \geq w(f)$. In view of our choice of e, we conclude $w(f) = w(e)$, so that f is an edge of T having the required property.

4.3.5 The assertion is an immediate consequence of Exercise 4.3.6. A direct proof of the special case in question could proceed as follows. Suppose that G contains two distinct minimal spanning trees T and T'. Order the edges of T and T' according to increasing weight and assume that both trees have their first $k-1$ edges in common, whereas they differ in their respective k^{th} edges:

$$T = \{e_1, \ldots, e_{k-1}, e_k, \ldots, e_{n-1}\} \quad \text{and} \quad T' = \left\{e_1, \ldots, e_{k-1}, e'_k, \ldots, e'_{n-1}\right\},$$

where (without loss of generality)

$$w(e_1) < \ldots < w(e_{n-1}) \quad \text{and} \quad w(e_k) < w(e'_k) < \ldots < w(e'_{n-1}).$$

Adding the edge e_k to T' yields a cycle $C_{T'}(e_k)$; by Theorem 4.3.1, $w(e_k) \geq w(f)$ for all edges $f \in C_{T'}(e_k)$. As the weights of the edges are distinct, all edges $f \neq e_k$ of $C_{T'}(e_k)$ have to be contained among the first $k-1$ edges $e_1, \ldots e_{k-1}$ of T'. Hence T contains the cycle $C_{T'}(e_k)$, a contradiction.

4.3.6 Let T be any minimal spanning tree, so that T satisfies condition (4.1). Then T can be transformed into any fixed minimal spanning tree T' by a sequence of edge exchanges as in the proof of Theorem 4.3.1, where we obtained a third minimal spanning tree $T'' = (T' \setminus \{e'\}) \cup \{e\}$. Moreover, we had $w(e) = w(e')$, so that such an exchange always transforms T' into a tree T'' with the same weight sequence. Hence the induction argument given in the proof of Theorem 4.3.1 actually proves the additional assertion of the present exercise.

4.4.6 Perturb the weight function w by adding small constants to the weights of edges having the same weight under w. Clearly, this may be done in such a way that the resulting weight function w' assigns distinct weights to distinct edges, and that $w'(e) < w'(e')$ holds if and only if either $w(e) < w(e')$ or $w(e) = w(e')$, but e precedes e' under the specified tiebreaking rule. By Exercise 4.3.5, there is a unique minimal spanning tree with respect to w', which implies the assertion.

4.4.11 Order the edges of G according to increasing weight. As the algorithm of Kruskal constructs a minimal spanning tree T by selecting edges in this order (as far as possible), no spanning tree having a smaller maximum edge weight can exist. Moreover, any two minimal spanning trees have the

same weight sequence by Exercise 4.3.6. Hence any spanning tree T' satisfies $W(T') \geq W(T)$.

4.4.15 Assign weight 1 to all the edges, and apply the algorithm of Boruvka in this situation. Then we could choose an arbitrary edge e_u leaving a given connected component $U \in M$. In general, there will exist two connected components $U, U' \in M$ which can be connected by two different edges of G; then choosing these two edges as e_u and $e_{u'}$ would create a cycle.

4.4.16 A minimal spanning tree has weight $2 + 13 + 21 + 35 + 51 = 122$.

4.4.17 The proof of Theorem 4.3.1 shows that the subgraph of the minimal spanning trees (for a given weight function w) is connected. If we assign weight $w(e) = 1$ to all edges e, we see that this implies that the whole tree graph is connected.

4.5.5 The edges $e_{15}, e_{14}, e_{13}, e_{12}, e_{11}, e_{10}$ and e_8 form a maximal spanning tree (of weight $28 + 27 + 26 + 24 + 10 + 9 + 8 = 132$). The edges are given in the order in which the algorithm of Kruskal would find them.

4.5.6 The following characterization of maximal spanning trees follows from Theorem 4.3.1 by replacing w by $-w$: a spanning tree T is maximal if and only if the condition

$$(*) \qquad w(e) \leq w(f) \quad \text{for all edges } f \text{ in } C_T(e)$$

holds for each edge $e \notin T$. Now let $e = uv$ be an edge of G not contained in T. By hypothesis, the unique path P from u to v in T has capacity $w(P) \geq w(e)$; this implies $(*)$ in view of $C_T(e) = P \cup \{e\}$, which proves the assertion.

4.5.8 The digraph shown in Figure B.12 provides an example.

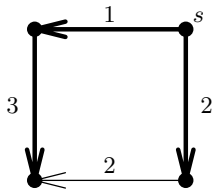


Fig. B.12. A digraph

4.7.7 Let T be an arbitrary spanning tree for G, and let x be a center of T. Denote the eccentricity of x in T by $e_T(x)$; then T has diameter either $d_T = 2e_T(x)$ or $d_T = 2e_T(x) - 1$ by Exercise 4.1.3. Clearly, x has eccentricity

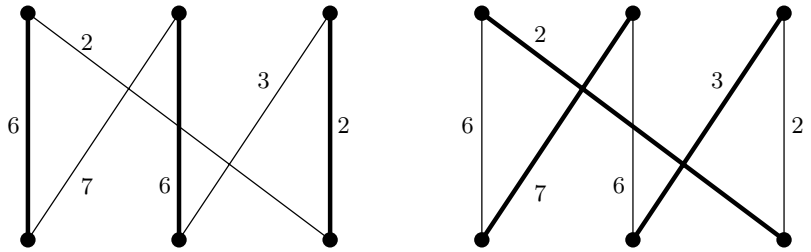
at most $e_T(x)$ in G. Thus it is an obvious approach to look for spanning trees whose centers are centers of G as well.

Now let z be a center of G, and let T_z be a spanning tree for G determined by a BFS starting at z. Note that T_z is an SP-tree for G with root z. It is easy to see that z is also a center of T_z. Therefore T_z has diameter $d = 2e$ or $d = 2e - 1$, where e denotes the eccentricity of z in G. Moreover, every other spanning tree has diameter at least $2e - 1$. Hence the tree T_z solves our problem; note that a center z (and then a tree T_z) can be determined with complexity $O(|V|^3)$ by Theorem 3.9.8.

We mention that it is easy to find examples where a BFS starting at z could either find a tree of diameter $2e$ or a tree of diameter $2e - 1$, depending on the order in which adjacent vertices are examined.

B.5 Solutions for Chapter 5

5.1.5 The network in Figure B.13 has a maximal matching of weight 14, but the greedy algorithm constructs a matching of weight 12.



maximal matching greedy matching

Fig. B.13. Two matchings

5.2.3 Let N be the incidence matrix of the graph $G = (V, E)$. Identify E with the set of columns of N and apply Theorem 4.2.3.

5.2.4 Let $A \subseteq E$. As the forests of G form the graphic matroid $M(G) = M_0(G)$ on E, A has a well-defined rank $\varrho(A)$ in $M(G)$, namely the maximal cardinality of a forest contained in A. We use this fact to verify condition (3) in Theorem 5.2.1 for $M = M_k(G)$, which will establish that M is likewise a matroid. We distinguish two cases: if $|A| - \varrho(A) \leq k$, then A itself is independent in M (and thus the only maximal independent set contained in A); and if $|A| - \varrho(A) > k$, then the maximal independent sets of M contained in A are the maximal forests in A enlarged by any k further edges from A (and thus all have cardinality $\varrho(A) + k$).

5.2.5 Conditions (1) and (2) are clear. To show that (3) holds, let J be a maximal independent subset of $A \cap B$, and choose a maximal independent subset K of $A \cup B$ containing J. Write $K = J \,\dot\cup\, X \,\dot\cup\, Y$ with $X \subset A$ and $Y \subset B$. Then $J \cup X$ and $J \cup Y$ are independent subsets of A and B, respectively, so that

$$\rho(A \cup B) + \rho(A \cap B) = 2|J| + |X| + |Y| = |J \cup X| + |J \cup Y| \leq \rho(A) + \rho(B).$$

5.2.9 By Theorem 5.2.8, $\sigma(X) = \{e \in E : \rho(X \cup \{e\}) = \rho(X)\}$; therefore condition (1) is clear. To prove (2), let J be a maximal independent subset of Y, and choose a maximal independent subset K of X containing J. If $e \in \sigma(Y)$, then $e \in \sigma(X)$: otherwise $K \cup \{e\}$ would be independent, so that $J \cup \{e\}$ would be independent as well, contradicting $\rho(J \cup \{e\}) = \rho(J)$).

By Theorem 5.2.8, $\sigma(X)$ is the unique maximal set containing X such that $\rho(\sigma(X)) = \rho(X)$; now (3) is clear. To show (4), let J be a maximal independent subset of X (and hence of $\sigma(X)$). As $y \notin \sigma(X)$ and $y \in \sigma(X \cup \{x\})$, $J \cup \{x\}$ and $J \cup \{y\}$ have to be independent sets. Moreover, $\rho(X \cup \{x\}) = \rho(X \cup \{y\}) = \rho(X \cup \{x, y\})$. But this implies $x \in \sigma(X \cup \{y\})$.

5.2.10 Let B be a basis of the matroid $M = (E, \mathbf{S})$. As $\rho(B) = \rho(E)$, Theorem 5.2.8 yields $\rho(B) = E$, so that B is a generating set for M. Suppose that B is not minimal. Then then there exists a proper subset C of B such that $B \subset E = \sigma(C)$. But then $\rho(E) = |C| < |B|$, which contradicts the fact that B is independent.

Conversely, let D be a minimal generating set and A a maximal independent subset of D. Then $\rho(D) = |A|$ implies $D \subset \sigma(A)$ and (using Exercise 5.2.9) $E = \sigma(D) \subset \sigma(\sigma(A)) = \sigma(A)$. Hence A is a generating set of M, and the minimality of D implies $A = D$. Thus D is independent. Now $\sigma(D) = E$, so that $|D| = \rho(E)$; therefore D is a basis of M.

5.2.11 Let A and B be two closed sets in M. Then

$$\sigma(A \cap B) \subset \sigma(A) \cap \sigma(B) = A \cap B \subset \sigma(A \cap B),$$

so that $A \cap B$ is closed as well; this establishes (a).

To prove (b), let A be a closed set containing X. Then $\sigma(X) \subset \sigma(A) = A$. Thus $\sigma(X)$ is contained in the intersection of all closed sets containing X. Now (a) implies that $\sigma(X)$ coincides with this intersection.

Finally, suppose that the condition in (c) is violated for some $x \in E \setminus X$, so that $\rho(X \cup \{x\}) = \rho(X)$. Then $x \in \sigma(X)$, and X cannot be closed. The converse is similar.

5.2.12 Let $\{x_1, \ldots, x_r\}$ be a basis of $(E, \mathbf{S})$. The 2^r subsets of this basis have 2^r distinct spans.

5.2.16 Suppose that condition $(2')$ does not hold. Choose two cycles C and D and elements $x \in C \cap D$ and $y \in C \setminus D$ violating $(2')$, so that $|C \cup D|$ is

minimal among all counterexamples. In view of Theorem 5.2.15, there exists a cycle $F_1 \subset (C \cup D) \setminus \{x\}$ with $y \notin F_1$. Note that the set $F_1 \cap (D \setminus C)$ cannot be empty: otherwise F_1 would be a proper subset of C. Hence we may choose an element $z \in F_1 \cap (D \setminus C)$.

Now consider the cycles D and F_1 and the elements $z \in D \cap F_1$ and $x \in D \setminus F_1$. Note that $D \cup F_1$ is a proper subset of $C \cup D$, since $y \notin D \cup F_1$. By the minimality of our counterexample, there exists a cycle F_2 such that $x \in F_2 \subset (D \cup F_1) \setminus \{z\}$. Consider C, F_2, $x \in C \cap F_2$, and $y \in C \setminus F_2$. Again, the minimality of our counterexample applies: there exists a cycle F_3 such that $y \in F_3 \subset (C \cup F_2) \setminus \{x\}$. As $C \cup F_2$ is contained in $C \cup D$, we have obtained a contradiction.

5.3.6 By Theorem 5.3.1, $\rho(E \setminus A^*) = \rho^*(A^*) - |A^*| + \rho(E) = \rho(E)$, since A^* is independent in M^*. As A is an independent subset of $E \setminus A^*$, A can be extended to a maximal independent subset (in M) of $E \setminus A^*$; we denote this subset by B. Then $\rho(B) = \rho(E)$, so that B is a basis of M. Hence $B^* = E \setminus B$ is a basis of M^* containing A^*.

5.3.7 First let B be a basis of M. Suppose that C^* is a cocircuit which is disjoint to B. Then $E \setminus B$ contains the cocircuit C^* and is dependent in M^*, which contradicts Corollary 5.3.2. Now suppose that a subset X of B intersects each cocircuit. Then $E \setminus X$ cannot contain any circuit of M^*, so that $E \setminus X$ must be independent in M^*. As $E \setminus X$ contains the basis $E \setminus B$ of M^*, we conclude $X = B$. Thus the bases are the minimal sets intersecting each cocircuit. The converse is shown in a similar manner.

5.3.8 Suppose $C \cap C^* = \{e\}$. Then the disjoint sets $A = C \setminus \{e\}$ and $A^* = C^* \setminus \{e\}$ are independent in M and in M^*, respectively. By Exercise 5.3.6, A and A^* can be extended to bases B and B^* of M and M^*, respectively, and these bases are disjoint. Hence $E = B \cup B^*$. As C and C^* are dependent, e can be contained neither in B nor in B^*, a contradiction.

5.3.9 Let B be a basis of M containing $C \setminus \{x\}$. As $B^* = E \setminus B$ is a basis of M^*, $B^* \cup \{y\}$ has to contain a unique cocircuit C^* of M by Theorem 5.2.13. Obviously, y must be contained in C^*. Now $x \notin C^*$ would imply $|C \cap C^*| = 1$, contradicting Exercise 5.3.8. Thus $x, y \in C \cap C^*$, so that $C \cap C^* = \{x, y\}$.

5.4.6 It suffices to find a subset A of E and two maximal independent subsets D and D' of A such that $2|D'| = n|D|$. We may assume $V = \{1, \dots n\}$. Then $D = \{(i, i+1) : i = 1, \dots, n-1\}$, $D' = \{(i, j) : i, j = 1, \dots n \text{ and } i > j\}$ and $A = D \cup D'$ have the required property.

5.4.10 M is the intersection of the graphic matroid $M(G)$, the head-partition matroid of G, and the tail-partition matroid of G.

5.5.7 Suppose w does not have to satisfy the triangle inequality. Then we may, for instance, increase the weight of the edge of maximal weight in Example 5.5.6 by an arbitrary value and thus make the solution determined by the greedy algorithm arbitrarily poor.

5.6.3 Suppose that (CC) is violated by some $A \in \mathbf{S}$, elements $x, y \in \mathrm{ext}(A)$, and a set $X \subset E \setminus (A \cup \mathrm{ext}(A))$. Thus there exists a basis B such that $A \cup X \cup \{x\} \subset B$, whereas $A \cup X \cup \{y\}$ is not contained in any basis. Consider the following weight function for E:

$$w(z) \;=\; \begin{cases} 3 & \text{if } z \in A \\ 2 & \text{if } z \in X \\ 1 & \text{if } z = y \\ 0 & \text{otherwise.} \end{cases}$$

Then the basis B has weight $w(B) = 3|A| + 2|X|$. The greedy algorithm begins by constructing (in some order) the feasible set A and then adds y; note that the elements of X have larger weight than y, but are not contained in $\mathrm{ext}(A)$. After that, the algorithm can add at most $|X| - 1$ of the elements of X, because we assumed that $A \cup X \cup \{y\}$ is not contained in any feasible set. Thus the solution generated by the greedy algorithm has weight at most

$$3|A| + 1 + 2(|X| - 1) \;<\; w(B),$$

a contradiction.

B.6 Solutions for Chapter 6

6.1.9 Replace each vertex v by a pair (v, v') of vertices, and each edge vw by $v'w$. Furthermore, add all edges of the form vv', and put $c(v'w) = c(vw)$ and $c(vv') = d(v)$. It is easily checked that a flow f' on the new network corresponds to a flow f on N satisfying (F3).

Now let (S, T) be a cut in the new network, and denote the set of edges e with $e^- \in S$ and $e^+ \in T$ by E'. Each edge of type $v'w$ corresponds to an edge vw in N, and each edge of type vv' corresponds to a vertex v of N. Thus the set E' of edges of the cut (S, T) corresponds to a cut in N in the following sense: a (generalized) *cut* is a set of edges and vertices (distinct from s and t) of G so that every directed path from s to t contains at least one of these edges and vertices. The *capacity* of such a cut is the sum of all $c(e)$ and $d(v)$ for edges e and vertices v, respectively, which are contained in the cut. Then the generalization of Theorem 6.1.6 states that he minimal capacity of a generalized cut equals the maximal value of a flow satisfying (F3). This theorem is easily derived by applying Theorem 6.1.6 to the network defined above.

6.1.10 If we require k vertices $s_1, \ldots, s_k$ as sources (so that (F2) does not have to be satisfied for these vertices, and as much flow as possible should originate there), we can add a new source s and all edges ss_i $(i = 1, \ldots k)$ with sufficiently large capacity.

6.1.11 Let W be the maximal value of a flow on N, and let (S, T) be a minimal cut; by hypothesis, $c(S, T) = W \neq 0$. If we remove an edge e with $e^- \in S$ and $e^+ \in T$ and $c(e) \neq 0$ from G, the capacity $c(S, T)$ and hence the value of a maximal flow is decreased by $c(e)$. This suggests to choose e as an edge of maximal capacity in a minimal cut. However, these edges do not have to be most vital, as the example of the network given in Figure 6.12 shows: here the edge sa is obviously most vital, but it is not contained in a minimal cut.

6.1.12 No: the flow in the flow network of Figure B.14 provides a counter-example.

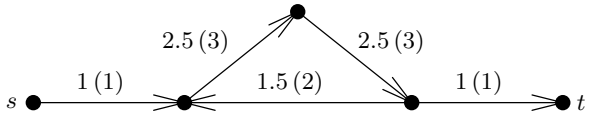


Fig. B.14. A flow

6.1.13 The capacities in the flow network of Figure B.14 actually define an integral flow, which is obviously maximal but not the sum of elementary flows.

6.1.14 First, in step (3) of Algorithm 6.1.7, we set $d(v) = 0$ for $v \neq s$. During the following labelling process, the labels are not permanent; similarly to the algorithm of Dijkstra, the label of the vertex v which is chosen in step (5) is made permanent at this point. As we want to construct augmenting paths of maximal capacity from s to all the other vertices, we choose in step (5) – among all labelled vertices v with $u(v) =$ false (that is, v is not yet permanent) – the vertex v for which $d(v)$ is maximal; initially, this is s.

Moreover, we do not change the flow as soon as t is reached, but wait until t is chosen in step (5) (and thus made permanent). For this purpose, we insert an **if** clause after step (5): if $v = t$, we may change the flow as in steps (16) to (28) of Algorithm 6.1.7; of course, we have to set $d(v) = 0$ for $v \neq s$ in step (27). Otherwise (if $v \neq t$), the labelling process is continued from v. As in steps (6) to (9), we first consider all edges of the form $e = vw$. If $u(w) =$ false (that is, w is not yet labelled permanently) and $d(w) < \min \{d(v), c(e) - f(e)\}$, then $d(w)$ is replaced by this minimum and w is labelled with $(v, +, d(w))$, so that the former label is also replaced. Steps (10) to (13) (for edges of the form $e = wv$) are changed in an analogous manner. Next v

is made permanent in step (14). We leave the details and the task of writing down a formal version of this method to the reader.

6.1.15 Let us write $P = S \cap S'$, $Q = T \cap S'$, $R = S \cap T'$, and $U = T \cap T'$; see Figure B.15. Denote the maximal flow value on N by W. By hypothesis,

$$W \;=\; c(S,T) \;=\; c(P,Q) + c(P,U) + c(R,Q) + c(R,U)$$

and

$$W \;=\; c(S',T') \;=\; c(P,R) + c(P,U) + c(Q,R) + c(Q,U).$$

On the other hand, using Lemma 6.1.2,

$$W \;\leq\; c(S \cap S', T \cup T') \;=\; c(P,Q) + c(P,R) + c(P,U)$$

and

$$W \;\leq\; c(S \cup S', T \cap T') \;=\; c(P,U) + c(Q,U) + c(R,U).$$

Hence

$$2W \;=\; c(S,T) + c(S',T') \;\geq\; c(S \cap S', T \cup T') + c(S \cup S', T \cap T') \;\geq\; 2W.$$

Thus we have equality throughout, implying

$$c(S \cap S', T \cup T') \;=\; c(S \cup S', T \cap T') \;=\; W.$$

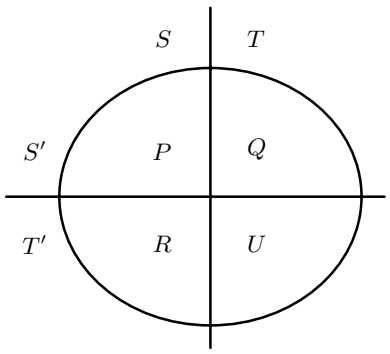


Fig. B.15. The cuts in Exercise 6.1.15

6.1.16 Let (S,T) be any minimal cut, and assume that some vertex $v \in T$ is accessible from s on an augmenting path P with respect to the given maximal flow f. Clearly, P has to contain an edge e which lies in the cocycle $E(S,T)$. If e is a forward edge, it cannot be saturated; and if e is a backward edge, it cannot be void. But this contradicts the characterization of minimal cuts given in Lemma 6.1.2, and we conclude $S_f \subseteq S$ for each minimal cut (S,T).

Hence the intersection S_0 of all such S contains S_f. By Exercise 6.1.15, S_0 is itself the s-part of a minimal cut, which proves the assertion: $S_f = S_0$.

6.2.4 A second maximal flow g can be obtained from the flow f_9 in Figure B.14 by letting the edge ct not carry any flow, and enlarging the value of the flow on the edges cf and ft accordingly: $g(ct) = 0$, $g(cf) = 15$, $g(ft) = 17$. Actually, there are further maximal flows, as there are several ways of distributing the flow emanating from c.

In contrast, (S_f, T_f) is the unique minimal cut, as can be seen using Exercise 6.1.16 and the criterion in Lemma 6.1.2. For instance, if we wanted to move the vertex b from the t-part T_f to the s-part S_f of the cut, we would have to include also c into the s-part, as the edge bc is not saturated. Conversely, if we wanted to include c, we also would have to include b, as bc is not void either. Using this type of argument shows that the s-part S_f cannot be enlarged at all.

6.2.5 We use the algorithm described in the solution to Exercise 6.1.14. During the first iteration, the vertices chosen in step (5) are s with $d(s) = \infty$, a with $d(a) = 38$, d with $d(d) = 13$, c with $d(c) = 10$, f with $d(f) = 10$, and t with $d(t) = 10$ (in this order). This yields an augmenting path with capacity 10; we obtain the flow f_1 of value 10 shown in Figure B.16, which also gives the labels determined by the first iteration.

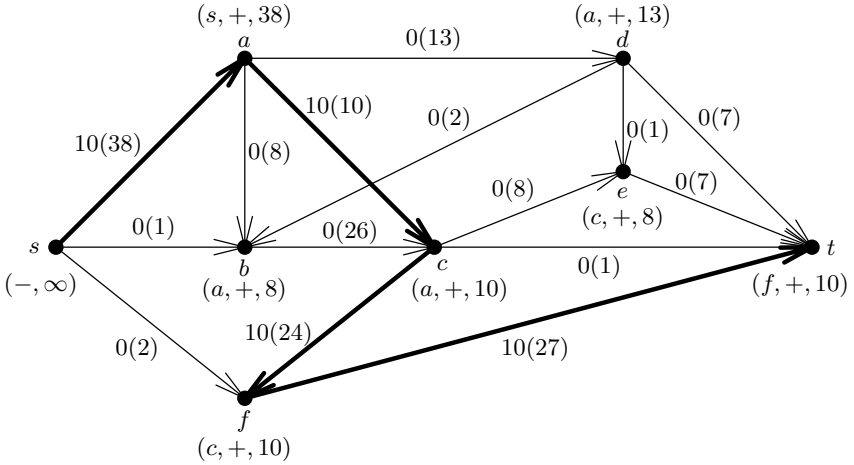


Fig. B.16. $w(f_1) = 10$

During the next iteration, the vertices s with $d(s) = \infty$, a with $d(a) = 28$, d with $d(d) = 13$, b with $d(b) = 8$, c with $d(c) = 8$, f with $d(f) = 8$, and t with $d(t) = 8$ are chosen in step (5). The corresponding augmenting path with capacity 8 yields the flow f_2 shown in Figure B.17.

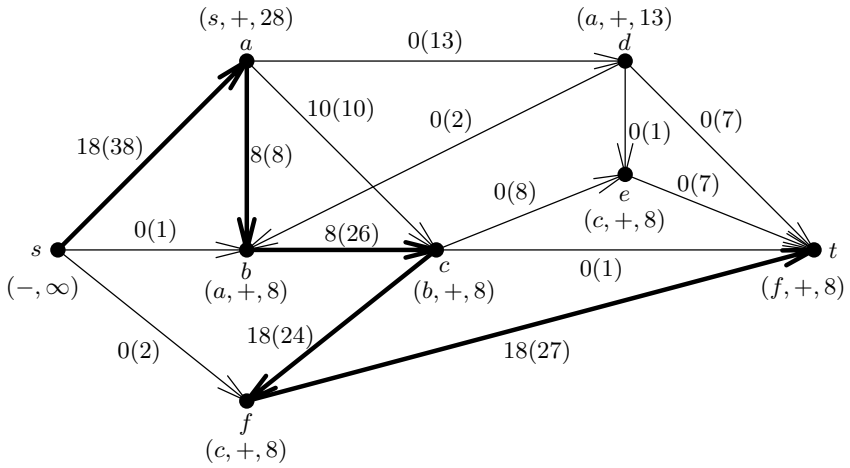


Fig. B.17. $w(f_2) = 18$

During the following iteration, the vertices chosen in step (5) are s with $d(s) = \infty$, a with $d(a) = 20$, d with $d(d) = 13$, and t with $d(t) = 7$. We obtain an augmenting path with capacity 7 and the flow f_3 shown in Figure B.18.

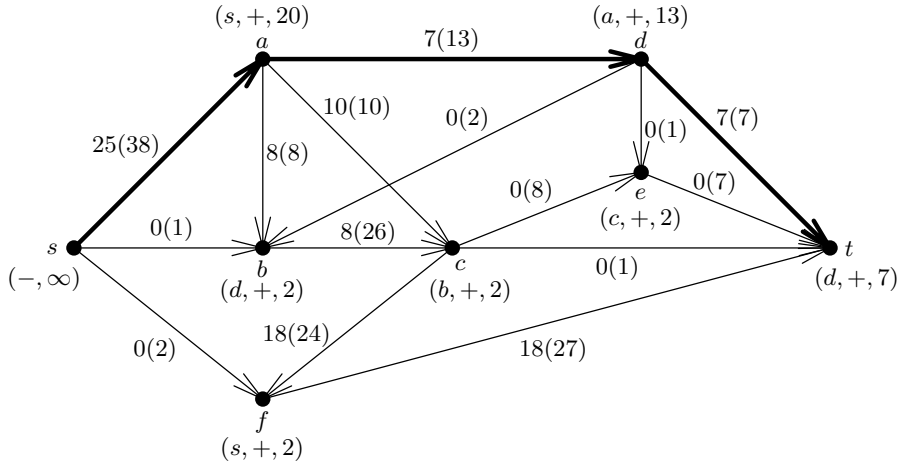


Fig. B.18. $w(f_3) = 25$

Four more iterations are needed; the augmenting paths constructed are

- $s - f - t$ with capacity 2,
- $s - a - d - b - c - f - t$ with capacity 2,
- $s - b - c - t$ with capacity 1,
- and $s - a - d - e - t$ with capacity 1.

The resulting flow f with $w(f) = 31$ is shown in Figure B.19.

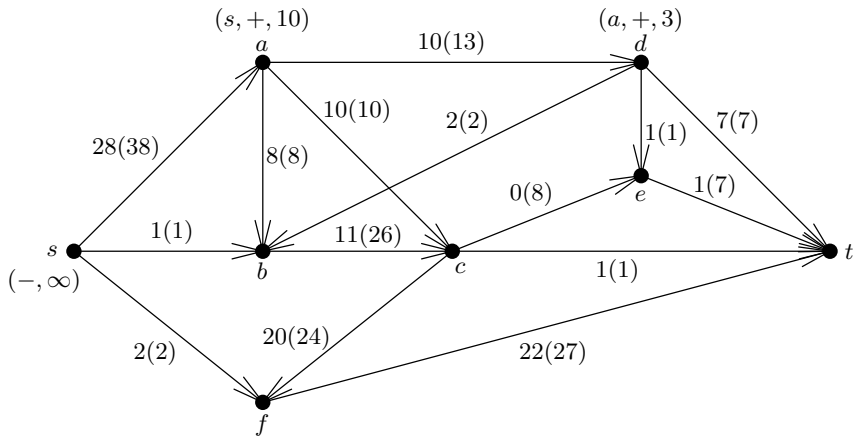


Fig. B.19. $w(f) = 31$

Thus this algorithm needs seven flow changes, whereas the algorithm of Edmonds and Karp used in Example 6.2.3 made nine changes. However, in the algorithm used here, the labelling process is somewhat more involved. Note that the maximal flows of Figures 6.12 and B.19 are not identical.

6.2.6 The maximal value of a flow is 5; Figure B.20 shows a flow f with $w(f) = 5$ and a cut having this capacity.

6.2.7 Let f be the flow of value $W = w(f)$ which was found for the incorrect capacity $d(e)$, let (S, T) be a minimal cut, and denote the correct capacity by $c(e)$. The results for the incorrect input data can be used when calculating a flow for the correct capacity as follows, where we distinguish two cases.

Case 1. $c(e) < d(e)$. It is clear that (S, T) is still a minimal cut, if e is contained in (S, T) (that is, $e^- \in S$ and $e^+ \in T$). In the corrected network, (S, T) has capacity $c(S, T) - (d(e) - c(e))$, so that the maximal value of a flow is $W' = W - (d(e) - c(e))$. To find a flow of value W', consider all the augmenting paths (constructed before) containing e and decrease the value of the corresponding flow by $d(e) - c(e)$.

If e is not contained in (S, T) and $f(e) \leq c(e)$, there is obviously nothing to change. If $f(e) > c(e)$, we decrease the flow by $f(e) - c(e)$ (as before) and run the algorithm again, using the decreased flow as the initial flow.

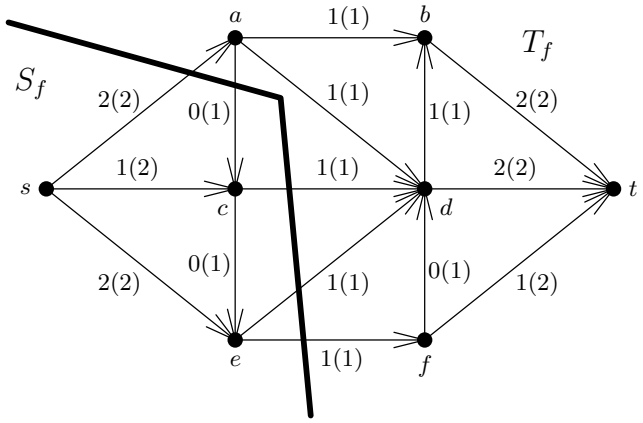


Fig. B.20. Solution to Exercise 6.2.5

Case 2. $c(e) > d(e)$: If e is not contained in (S,T), then (S,T) is still a minimal cut and there is nothing to change. Otherwise, we run the algorithm again, using f as the initial flow.

6.2.8 Note that the edge $e = ac$ is contained in the minimal cut (S,T) shown in Figure 6.12. If $c(e) = 8$, (S,T) is still a minimal cut, so that the value of the flow has to be decreased to 29. A maximal flow of this value can be constructed from the flow of Figure 6.12 by decreasing the flow values of all edges in the augmenting path shown in Figure 6.7 by 2. For $c(e) = 12$, the same augmenting path can be used for increasing the value of the flow to 33.

6.2.9 First, the capacity of ac is increased to 12, so that the value of the flow can be increased to 33 (by increasing $f(e)$ by 2 for each of the edges $e = sa, ac, cf, ft$); see Exercise 6.2.8. Since the edge ad is not contained in the minimal cut (S,T), increasing the capacity of this edge does not affect the maximal flow. Now we delete the edge de. As this edge is contained in (S,T), the value of the flow has to be decreased by 1, say along the path $s - a - d - e - t$. Finally, ct is removed. The value of a maximal flow is not changed, because the unit of flow carried by ct can be moved along the path $c - f - t$ instead. We obtain the flow of value 32 shown in Figure B.21; note that (S_f, T_f) is still a minimal cut, as it should be according to Exercise 6.1.16.

6.3.5 By definition, $c'(S,T)$ is the sum of all $c'(x)$ for $x^- \in S$ and $x^+ \in T$. If $x = e'$ corresponds to a forward edge e, we have $c'(x) = c(e) - f(e)$. Otherwise (if $x = e''$ corresponds to a backward edge e), $c'(x) = f(e)$. Thus

$$c'(S,T) = \sum_{e^- \in S, e^+ \in T} c(e) - \sum_{e^- \in S, e^+ \in T} f(e) + \sum_{e^- \in T, e^+ \in S} f(e);$$

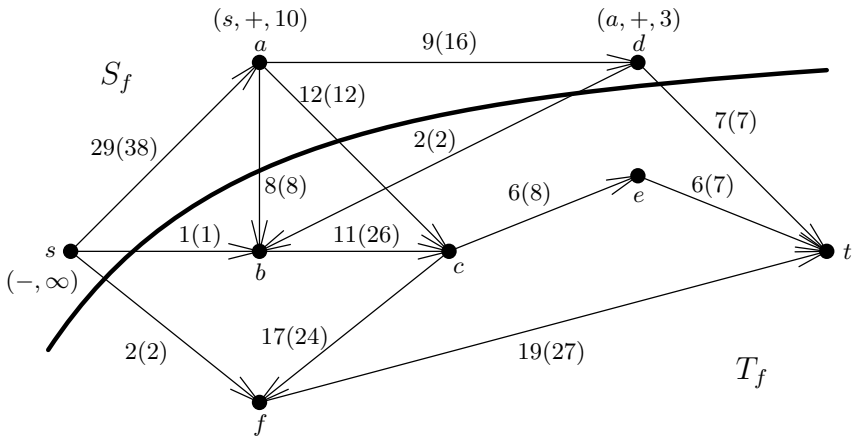


Fig. B.21. $w(f) = 32 = c(S, T)$

hence, using Lemma 6.1.2, $c'(S, T) = c(S, T) - w(f)$. In particular, this holds for minimal cuts, and the assertion follows by applying Theorem 6.1.6 to both networks.

6.3.8 Execute a BFS starting at t on the digraph with opposite orientation, and remove all vertices which are not reached during the algorithm.

6.3.9 The network N'' and a blocking flow are shown in Figure B.22.

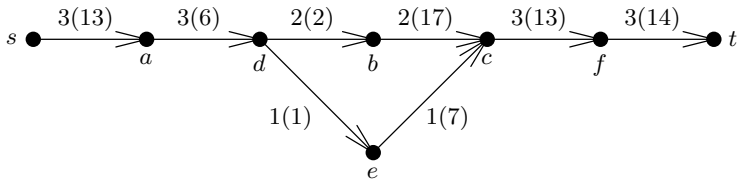


Fig. B.22. A blocking flow

6.3.10 Consider Example 6.3.7, and note that the blocking flow g on $N''(f)$ of value 10 leads to a maximal flow g' of value 11 on $N''(f)$. The underlying flow f has value 10, whereas the maximal value of a flow on N is $31 \neq 10+11$; see Example 6.2.3.

6.3.13 The layered auxiliary network with respect to g on $N'(f)$ is shown in Figure B.23, and the layered auxiliary network with respect to g on $N''(f)$ is shown in Figure B.24. The flow determined on N by f and g is the flow $h = f_6$ shown in Figure 6.9. Thus $N''(h)$ is equal to the network of Figure B.23.

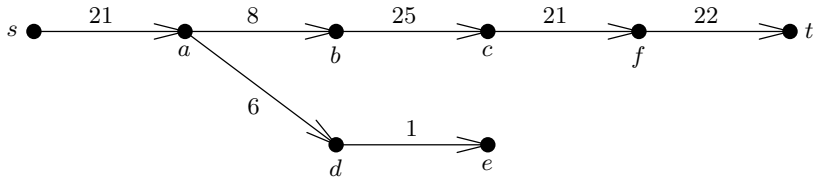


Fig. B.23. Layered auxiliary network for $N'(f)$

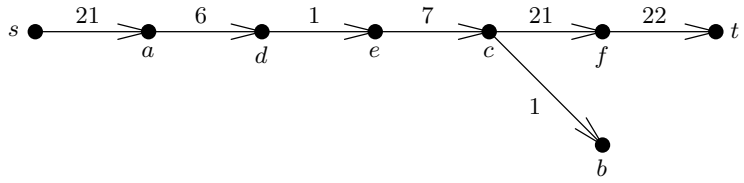


Fig. B.24. Layered auxiliary network for $N''(f)$

6.3.19 Replace step (17) of procedure AUXNET (Algorithm 6.3.14) by

(17′) **if** $t \in V''$ **then** max ← false; $d \leftarrow i$ **else** max ← true;
$S \leftarrow V''$; $T \leftarrow V \setminus S$ **fi**

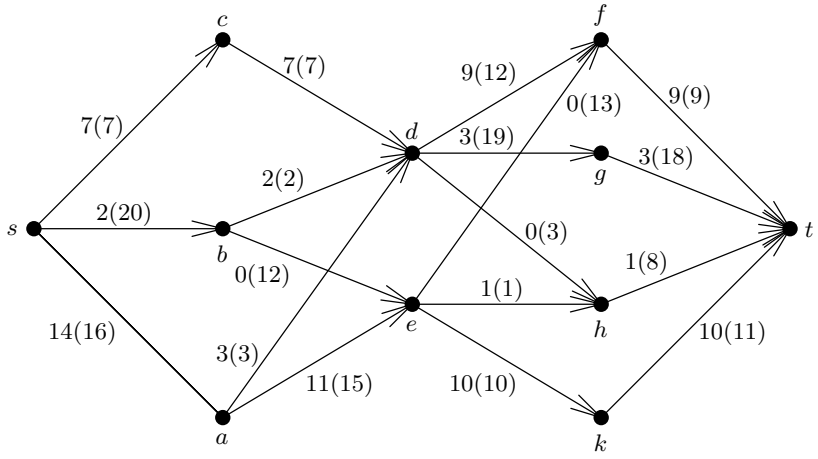


Fig. B.25. Solution to Exercise 6.4.5

6.4.5 A blocking flow determined by Algorithm 6.4.1 is shown in Figure B.25. The paths corresponding to the sequences (s, a, d, f, t), (s, b, d, f, t), (s, c, d, f, t), (s, c, d, g, t), (s, a, e, h, t), (s, a, e, k, t) were constructed in this

order (as usual, if there were several possible ways of choosing the edge $e = uv$ in step (5), we have proceeded according to the alphabetical order of the vertices); their capacities are 3, 2, 4, 3, 1, and 10, respectively. Thus the total value of the flow is 23.

6.4.10 Algorithm 6.4.6 needs four iterations, where the vertices of minimal potential are h with $p(h) = 4$, c with $p(c) = 7$, d with $p(d) = 2$, and e with $p(e) = 10$, respectively. The resulting blocking flow of value 23 is shown in Figure B.26. Note that it is not identical with the one given in Figure B.25.

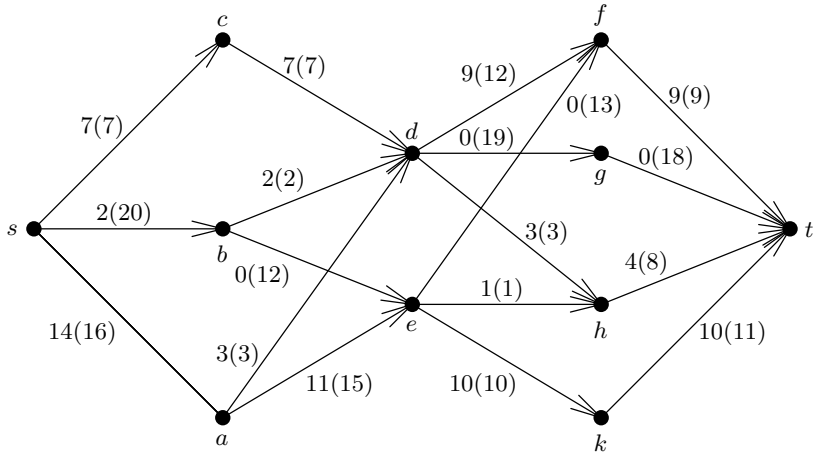


Fig. B.26. Solution to Exercise 6.4.10

6.5.6 Define a bipartite graph G on $S \dot\cup T$, where $S = \{1, \ldots, m\}$ and $T = \{1', \ldots, n'\}$, and let $\{i, j'\}$ be an edge if and only if girl i and boy j' know each other. Then the desired arrangement for a dance obviously corresponds to a matching of maximal cardinality in G; a solution can be determined using Example 6.5.5.

6.5.7 Let $A \subset S$, and let $X \subset A$ be an independent subset of maximal cardinality of A, say $|X| = k$. Consider the network N constructed from G in Example 6.5.5. Remove all vertices of $S \setminus A$ together with all edges incident with them from N, and denote the resulting network by N_A. Moreover, let M be a matching of G with $X = \{e^- : e \in M\}$. As we saw in Example 6.5.5, M induces a flow of value k on N_A.

Now let Y be a maximal independent subset of A, say $Y = \{e^- : e \in M'\}$ for some matching M'; by hypothesis, $|Y| \leq k$. Suppose $|Y| < k$. Then the flow on N_A corresponding to M' cannot be maximal, and a maximal flow f can be obtained by constructing $k - |Y|$ augmenting paths in N_A. It is easy to see that there always is a matching corresponding to an independent

subset of A containing Y (for each change of the flow). Thus Y cannot have been maximal either, a contradiction. Hence any two maximal independent subsets of A have the same cardinality k, so that $(S, \mathbf{S})$ satisfies condition (3) of Theorem 5.2.1 and therefore is a matroid.

Such matroids are called *transversal matroids*; they are considered in Section 7.3. We have given an algorithmic proof for the fact that $(S, \mathbf{S})$ is a matroid, by showing in a constructive way that condition (3) of Theorem 5.2.1 is satisfied. In a similar manner, the validity of condition (3) can be proved also – in the language of transversal theory – by using the algorithm of [Hal56]; see Section 7.3.

6.6.13 As there are only $n-2$ vertices distinct from s and t, at least one of the numbers i in the range $1 \leq i \leq n-1$ does not occur as a label $d(v)$. Choose such an i, and consider $S = \{v \in V : d(v) > i\}$ and $T = \{w \in V : d(w) < i\}$. Note $s \in S$ and $t \in T$. Our selection of i implies $d(v) \geq d(w)+2$ for all choices of $v \in S$ and $w \in T$. Thus no edge $e = vw$ with $v \in S$ and $w \in T$ can belong to the residual graph G_f, since it violates the condition $d(v) \leq d(w) + 1$. As explained at the beginning of Section 6.6, G_f corresponds to the auxiliary network $N'(f)$ used in the classical algorithms. Using similar arguments, it is easily seen that the fact that no edge $e = vw$ with $v \in S$ and $w \in T$ belongs to G_f translates into the statement that each edge e with $e^- \in S$ and $e^+ \in T$ is saturated, whereas each edge e with $e^- \in T$ and $e^+ \in S$ is void. Now Lemma 6.1.2 shows that (S, T) is a minimal cut.

6.6.19 The algorithm FIFOFLOW determines (after nine phases) the maximal flow shown in Figure 6.12. It needs four more RELABEL- and five more PUSH-operations than HLFLOW – that is, about one third more operations altogether.

B.7 Solutions for Chapter 7

7.1.3 We use the algorithm of Edmonds and Karp. Suppose that there exists an augmenting path containing a backward edge, and let P be the first such path and $e = uv$ be the last backward edge in P. When P is constructed, we must have $f(e) \neq 0$. Let Q be the last augmenting path constructed before P for which $f(e)$ was changed (and actually increased). Then P and Q have the form

$$P: \quad s \ \overline{\quad P' \quad} \ v \ — \ u \ \overline{\quad P'' \quad} \ t$$

and

$$Q: \quad s \ \overline{\quad Q' \quad} \ u \ — \ v \ \overline{\quad Q'' \quad} \ t.$$

Denote the capacities of P and Q by γ and δ, respectively. Suppose first that $\gamma \leq \delta$. Then we may replace Q and P by the following three paths:

$s \overset{Q'}{\text{---}} u \text{ --- } v \overset{Q''}{\text{---}} t$ (with capacity $\delta - \gamma$);

$s \overset{Q'}{\text{---}} u \overset{P''}{\text{---}} t$ (with capacity γ);

$s \overset{P'}{\text{---}} v \overset{Q''}{\text{---}} t$ (with capacity γ).

Then P'', Q', and Q'' contain only forward edges, and the sum of the capacities of these three paths is $\gamma + \delta$, so that we have removed the backward edge e from P.

For $\gamma > \delta$, we use similar arguments to replace P and Q by three paths whose capacities sum to $\gamma + \delta$. However, the backward edge e is not removed in this case, since we need the path P with capacity $\gamma - \delta$. Nevertheless, the capacity of P is decreased, so that this method has to terminate. As the algorithm of Edmonds and Karp is finite, we get a finite method for constructing a maximal flow which uses only augmenting paths consisting exclusively of forward edges.

In Example 6.2.3, the only backward edge occurs in the last augmenting path, which has capacity 1 (see Figure 6.11):

$$P: \quad s \text{ --- } a \text{ --- } d \text{ --- } e \text{ --- } c \text{ --- } f \text{ --- } t;$$

the backward edge is ce. Here Q is the following augmenting path:

$$Q: \quad s \text{ --- } a \text{ --- } c \text{ --- } e \text{ --- } t,$$

which has capacity 7; see Figure 6.6. As described above, we may replace P and Q by the following three augmenting paths:

- $s - a - c - e - t$ (with capacity 6);
- $s - a - c - f - t$ (with capacity 1);
- $s - a - d - e - t$ (with capacity 1).

7.1.7 Clearly, the proposed criterion is sufficient. Now let G be k-connected. By Menger's theorem, any two non-adjacent vertices of G are connected by k vertex disjoint paths. It remains to consider adjacent vertices s and t. Let H be the graph obtained by removing the edge st from G. Obviously, H is at least $(k-1)$-connected. Again by Menger's theorem, s and t are connected in H by $k-1$ vertex disjoint paths. Then st is the k-th path from s to t in G.

7.1.8 By Exercise 7.1.7, any two vertices of a k-connected graph are connected by k vertex disjoint paths, so that every vertex must have degree at least k. On the other hand, Exercise 1.5.14 shows that a planar graph has to contain vertices of degree at most 5. This proves the first assertion.

The graph with six vertices shown in Figure B.5 is 4-connected. If G is 5-connected, every vertex must have degree at least 5. As in the solution to

Exercise 1.5.14, we get the following bound on the number n_5 of vertices of degree at most (and hence equal to) 5:

$$6(n - n_5) + 5n_5 \ \leq \ 12n - 6;$$

thus $n \geq n_5 \geq 12$. The icosahedral graph provides an example with twelve vertices; see Figure 9.1.

7.1.9 Add two vertices s and t and all edges sx for $x \in S$ as well as all edges yt for $y \in T$ to G. Then the assertion follows from Theorem 7.1.4.

7.1.10 Let G be the given graph, and s and t the specified vertices. Consider the graph H whose vertices are the edges of G together with s and t and define adjacency as follows: two edges of G are adjacent in H if and only if they share a common vertex $v \neq s, t$ in G; any edge e of the form sv is adjacent to s; and any edge e of the form vt is adjacent to t. Note that s and t are not adjacent in H.

It is clear that edge disjoint paths from s to t in G are transformed into vertex disjoint paths in H by the preceding construction. In the converse direction, one has to be a bit careful, as a path in H corresponds to a trail in G, but not necessarily to a path. However, this difficulty can be overcome by appealing to Exercise 1.2.1, which guarantees that we may select a path contained in a given trail.

Finally, it is again clear that edge separators for s and t in G correspond to vertex separators for s and t in H. Hence the undirected case of Theorem 7.1.1 indeed reduces to the undirected case of Theorem 7.1.4. Note that the same approach also works in the directed case; here one needs to use Exercise 1.6.6 instead of Exercise 1.2.1.

7.2.2 An unextendable matching M' has at least $k/2$ edges: otherwise, at least one of the k edges of a maximal matching M could be added to M'. It is easy to construct examples which show that this bound is best possible.

7.2.4 As explained in Example 6.5.5, we may run the labelling algorithm (or, more efficiently, Dinic' algorithm) to determine a maximal 0-1-flow f and hence a maximal matching M. Let us denote the associated minimal cut by (X, Y), where X consists of all vertices which are accessible from s on an augmenting path. We write S_X for $S \cap X$, and S_Y for $S \cap Y$; the analogous subsets of T will be denoted by T_X and T_Y.

We claim that $W = S_Y \cup T_X$ is a minimum cardinality vertex cover for G. The only edges which might not be covered by W are the edges of the form vw with $v \in S_X$ and $w \in T_Y$. If such an edge vw does not belong to M, it does not carry any flow, and hence could be used to extend an augmenting path from s to v (which exists, as $v \in S_X$) on to w, contradicting $w \in T_Y$. It remains to consider the case where $vw \in M$. But then the edge sv is saturated, so that v can only be reached via an augmenting path from s whose final edge is vw,

used as a backward edge; again, this gives the contradiction $w \in T_X$. Thus W is indeed a vertex cover.

Finally, we show that W and M have the same cardinality. By the max-flow min-cut theorem, it suffices to check $|W| = c(X, Y)$, since $|M|$ equals the value of the maximal flow f. As all capacities are 1, we simply need to count the edges in the cocycle $C = E(X, Y)$. Trivially, an edge sv belongs to C if and only if $v \notin S_X$, giving $|S_Y|$ edges in C with start vertex s. Similarly, we get $|T_X|$ edges in C with start vertex in T_X (and end vertex t). Altogether, we now already have $|W|$ edges in C, so that C should not contain any further edges; indeed, any edge vw with start vertex $v \in S_X$ necessarily has $w \in T_X$, as we have already seen when proving that W is a vertex cover.

7.2.8 The Petersen graph (see Figure 1.12) is 3-regular, but does not have a 1-factorization. Assume otherwise. Then at least one of the three 1-factors involved, say M, has to contain two edges of the outer cycle, say the two edges drawn as dashed lines in Figure B.27. But this already determines M uniquely: for instance, the fifth point of the outer circle forces M to contain the spoke edge through that point. Hence M is the 1-factor consisting of the five dashed edges in Figure B.27. But the complement of M is the union of two vertex disjoint 5-cycles, and thus cannot split into two 1-factors. (This argument is taken from [Vol04].)

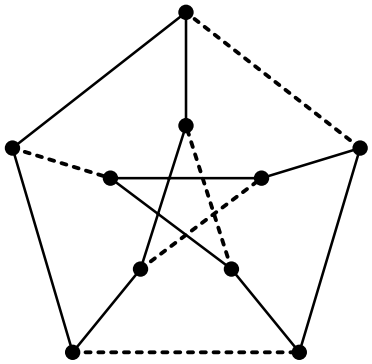


Fig. B.27. A 1-factor of the Petersen graph

7.2.10 Let us choose the disjoint union of the three $2n$-sets $R = \{r_1, \ldots, r_{2n}\}$, $S = \{s_1, \ldots, s_{2n}\}$, and $T = \{t_1, \ldots, t_{2n}\}$ as the vertex set of K_{6n}. Moreover denote the complete bipartite graph on $S \dot\cup T$ by K_{ST}, and the 1-factor $\{s_i t_i : i = 1, \ldots, 2n\}$ of K_{ST} by F_{ST}.

By Corollary 7.2.7 and Exercise 1.1.2, both $G_{ST} = K_{ST} \setminus F_{ST}$ and the complete graph K_R on R can be decomposed into $2n-1$ 1-factors. By choosing an arbitrary bijection between these two sets of 1-factors and by merging all

the corresponding factors, we obtain $2n-1$ 1-factors of K_{6n}; altogether, these factors contain precisely all the edges of one of the types s_it_j and r_ir_j (for $i \neq j$).

The same method yields (for the two cyclic permutations of the sets R, S, and T) $4n-2$ further 1-factors of K_{6n}. The remaining edges which do not occur in one of these $6n-3$ 1-factors are of the form r_is_i, r_it_i, and s_it_i (for $i = 1, \ldots, 2n$); obviously, these edges form a Δ-factor.

7.2.11 Denote the nine vertices by ij, where $i, j = 0, 1, 2$. Then the edges where i is constant form three triangles which yield a first Δ-factor; similarly, we obtain a second Δ-factor for constant j; then the remaining two Δ-factors are uniquely determined. This unique decomposition of K_9 into Δ-factors is – using geometric terminology – just the affine plane of order 3; see, for instance, [BeJL99].

7.2.12 Choose $2n-1$ factors of a 1-factorization of K_{6n-2} (see Exercise 1.1.2) and denote the graph formed by these factors by G. Then $\overline{G}$ is regular with degree $(4n-2)$ and, hence, can be decomposed into 2-factors by Theorem 7.2.9. Now choose a bijection between these two sets of $2n-1$ factors and merge corresponding factors.

7.3.2 The assertion is clear for $n = 1$. Thus let $n > 1$. Choose $x_1 \in A_1$ and put

$$\mathbf{B} \;=\; (B_2, \ldots, B_n) \quad \text{with} \quad B_i \;=\; A_i \setminus \{x_1\}.$$

Assume first that $\mathbf{A}$ does not contain a critical subfamily. Then the union of any k sets in $\mathbf{A}$ contains at least $k+1$ elements; thus $\mathbf{B}$ clearly satisfies (H′). Hence $\mathbf{B}$ contains a transversal T, so that $T \cup \{x_1\}$ is a transversal of $\mathbf{A}$.

It remains to consider the case where $\mathbf{A}$ contains a critical subfamily, say $\mathbf{A}' = (A_1, \ldots, A_m)$. By the induction hypothesis, $\mathbf{A}'$ contains a transversal T'. Put $\mathbf{C} = (C_{m+1}, \ldots, C_n)$, where $C_i = A_i \setminus T'$. Now one checks that $\mathbf{C}$ likewise satisfies condition (H′), so that $\mathbf{C}$ has a transversal T''. Then $T' \cup T''$ is a transversal of $\mathbf{A}$.

7.3.3 It is obvious that the maximal cardinality of a matching of G cannot exceed the minimal cardinality of a vertex cover of G. Now suppose that $X = S' \cup T'$ (where $S' \subset S$ and $T' \subset T$) is a minimal vertex cover. We will apply Theorem 7.3.1 in the terminology used in Theorem 7.2.5.

Consider the bipartite graph G' induced on the set $(S \setminus S') \,\dot\cup\, T'$. We want to show that G' satisfies condition (H). Suppose otherwise. Then there exists a subset J of T' with $|\Gamma(J)| < |J|$, so that the set $S' \cup \Gamma(J) \cup (T' \setminus J)$ is a vertex cover for G which has smaller cardinality than $|X|$. This contradicts our assumption above and proves that G' satisfies (H). By Theorem 7.2.5, G' has a matching of cardinality $|T'|$.

Similarly, the bipartite graph G'' induced on the set $S \cup (T \setminus T')$ contains a matching of cardinality $|S'|$. Then the union of these two matchings of G' and G'' forms a matching of cardinality $|X|$ of G.

7.3.6 The maximal cardinality of a matching in a bipartite graph (with vertex set $S \dot\cup T$) is $|T| - \max\{|J| - |\Gamma(J)| : J \subset T\}$.

7.3.10 Consider the family $\mathbf{A}$ which consists of d_i copies of A_i for $i = 1, \ldots, k$. Then $\mathbf{S}$ is precisely the set of partial transversals of $\mathbf{A}$, so that the assertion follows from Theorem 7.3.8.

7.3.13 Trivially, (1) follows from (2). So suppose that (1) holds. Write $m = |S|$ and assume $\mathbf{A}' = (A_1, \ldots, A_k)$. Let D be an arbitrary set of cardinality n which is disjoint to S, and consider the family $\mathbf{B}$ consisting of the sets

$$A_1, \ldots, A_k, A_{k+1} \cup D, \ldots, A_n \cup D \text{ and } m \text{ times the set } (S \setminus S') \cup D.$$

Now suppose that $\mathbf{B}$ has a transversal. As $\mathbf{B}$ consists of $m+n$ subsets of the set $S \cup D$ having $m+n$ elements, this transversal has to be $S \cup D$ itself. Thus, S is a transversal of a subfamily of $\mathbf{B}$ which contains all the sets $A_1, \ldots, A_k$, some of the sets $A_{k+1} \cup D, \ldots, A_n \cup D$, and some copies of $(S \setminus S') \cup D$. If we delete all those elements representing copies of $(S \setminus S') \cup D$ from S, we obtain a subset S'' of S which contains S' and is a transversal for a subfamily of $\mathbf{A}$ containing $\mathbf{A}'$.

It remains to show that the family $\mathbf{B}$ defined above satisfies condition (H′) of the marriage theorem. This condition is

$$\left| (\bigcup_{j \in J} A_j) \cup (\bigcup_{j \in K} A_j \cup D) \cup (\bigcup_{i=1}^{c} (S \setminus S') \cup D) \right| \;\geq\; |J| + |K| + c \qquad \text{(B.1)}$$

for all $J \subset \{1, \ldots, k\}$, $K \subset \{k+1, \ldots, n\}$ and $c \in \{0, \ldots, m\}$. First consider the case $c = 0$. If $K = \emptyset$, (B.1) follows from condition (H′) for $\mathbf{A}'$, which holds as $\mathbf{A}'$ has a transversal. If $K \neq \emptyset$, the union on the left hand side contains the n-set D, so that (B.1) is satisfied because of $n \geq |J| + |K|$. Now let $c \neq 0$; it suffices to consider the case $c = m$. As D and S are disjoint, (B.1) becomes

$$\left| \bigcup_{j \in J} A_j \cup (S \setminus S') \right| \;\geq\; |J| + m - n \quad \text{for } J \subset \{1, \ldots, n\}. \qquad \text{(B.2)}$$

But

$$\left| \bigcup_{j \in J} A_j \cup (S \setminus S') \right| \;=\; m - |S'| + \left| (\bigcup_{j \in J} A_j) \cap S' \right|,$$

so that (B.2) is equivalent to

$$\left| (\bigcup_{j\in J} A_j) \cap S' \right| \geq |J| + |S'| - n.$$

This condition holds by Theorem 7.3.7, as S' is a partial transversal of $\mathbf{A}$.

7.3.14 Let G be the bipartite graph with vertex set $S \dot{\cup} T$ corresponding to $\mathbf{A}$. As in Exercise 6.5.7, one sees that there is also a matroid induced on T. Using the terminology of set families, the independent sets of this matroid are precisely those subsets of the index set T for which the corresponding subfamily of $\mathbf{A}$ has a transversal.

7.3.18 Let $\mathbf{B}$ be the family consisting of p_i copies of A_i for $i = 1, \ldots, n$. Then the existence of sets X_i with the desired properties is equivalent to the existence of a transversal of $\mathbf{B}$. Now condition (H′) for $\mathbf{B}$ is precisely the condition given in the exercise, so that the assertion follows from the marriage theorem.

7.4.13 The assertions of Corollaries 7.4.6 and 7.2.7 are equivalent.

7.4.14 Let D be a diagonal with entries $d_1, \ldots, d_n$ satisfying $d_1 \ldots d_n \geq n^{-n}$. The inequality between the arithmetic and the geometric mean[3] implies

$$(d_1 \ldots d_n)^{1/n} \leq \frac{d_1 + \ldots + d_n}{n},$$

so that $d_1 + \ldots + d_n \geq 1$.

7.4.15 Let $\mathbf{T}$ be the set family as described in the hint. Then $\mathbf{T}$ satisfies condition (H′), since the ktr entries 1 in any given k rows of A have to be contained in at least kt columns of A (note that A has column sums $\leq r$). Therefore $\mathbf{T}$ has a transversal, so that there exist pairwise disjoint t-subsets S_i of T_i for $i = 1, \ldots, m$. Then the matrix P with entries $p_{ij} = 1$ for $i \in S_j$ and $p_{ij} = 0$ otherwise has row sums t and column sums ≤ 1. Moreover, the matrix $\mathbf{A}' = A - P$ has row sums $t(r-1)$.

As we want to use induction on r, we still have to make sure that the set X of all those indices for which column j of A has sum r is contained in $S_1 \cup \ldots \cup S_m$ (so that $\mathbf{A}'$ has column sums $\leq r-1$). By Corollary 7.3.9, it is sufficient to show that X is a partial transversal of $\mathbf{T}$. However, any k columns having sum r together contain precisely kr entries 1, and these entries have to be contained in at least k/t rows of A. As each T_i occurs precisely t times in $\mathbf{T}$, any k elements of X correspond to at least k sets in $\mathbf{T}$. Now Theorem 7.2.5 implies that X is a partial transversal.

[3] For a proof of the inequality mentioned above and of a more general inequality due to Muirhead [Mui03] using the methods of transversal theory, we refer the reader to [Mir71b, Theorem 4.3.3].

7.4.16 Using the equivalence of 0-1-matrices and bipartite graphs discussed at the beginning of Section 7.4, the assertion amounts to showing that a bipartite graph of maximal degree r can be decomposed into r matchings. Let $S \dot\cup T$ be the vertex set of G, and denote the set of vertices of degree r in S and T by S' and T', respectively. By Theorem 7.2.5, there exist matchings M' and M'' of G which meet S' and T', respectively. By Corollary 7.3.12, there also exists a matching M meeting $S' \cup T'$. Then $G \setminus M$ has maximal degree $r-1$, and the assertion follows by induction.

7.4.17 We may assume $n \geq 3$. We show first that the subspace W of $\mathbb{R}^{(n,n)}$ spanned by the permutation matrices consists precisely of those matrices for which all row and column sums are equal. Obviously, any linear combination of permutation matrices is contained in W and has constant row and column sum. Conversely, let A be a matrix with constant row and column sum. If A does not contain any negative entries, A is contained in W by Theorem 7.4.7. Otherwise, put $b = \max\{-a_{ij} : i,j = 1,\dots,n\}$. Then the matrix $B = A + bJ$ (where J is the matrix with all entries 1) has nonnegative entries and constant row and column sum. Therefore J and B (and A as well) are linear combinations of permutation matrices.

Now let W' be the subspace spanned by the $2n-2$ matrices S_i and Z_i (for $i = 1,\dots,n-1$) which have entry 1 in cell (n,i) and in cell (i,n), respectively, and all other entries 0. Obviously, W and W' have only the zero matrix in common. Thus $\dim W = n^2 - 2n + 2$ follows if we can show that W and W' together generate $\mathbb{R}^{(n,n)}$. Let A be an arbitrary matrix in $\mathbb{R}^{(n,n)}$. By adding appropriate multiples of S_i or of Z_i to A, we can obtain a matrix C for which the first $n-1$ rows and the first $n-1$ columns have a fixed sum s. Then the last row and the last column of C must have identical sum, say x. Adding aS_i and aZ_i to C, the sum s can be changed to $s' = s + a$; simultaneously, x is changed to $x' = x + (n-1)a$. As $n \neq 2$, we can determine a so that $x' = s'$; that is, the resulting matrix C' has constant row and column sum. Thus C' is contained in W, so that A is contained in $W + W'$.

7.5.4 Suppose G is a minimal counterexample to the assertion, and let $\mathbf{D}$ be a dissection of G consisting of as few paths as possible. Then $\mathbf{D}$ contains at least $\alpha + 1$ paths. Suppose we have $|\mathbf{D}| \geq \alpha + 2$. We omit a path W from $\mathbf{D}$. As G is minimal, $G \setminus W$ has a dissection into at most α paths, say $\mathbf{D}'$. But then $\mathbf{D}' \cup \{W\}$ is a dissection of G into $\alpha + 1$ paths contradicting our assumption.

Hence $|\mathbf{D}| = \alpha + 1$, say $\mathbf{D} = \{W_1,\dots,W_{\alpha+1}\}$. Denote the start vertex of W_i by p_i. By definition of α, the $\alpha+1$ vertices p_i cannot form an independent set; we may assume that p_1p_2 is an edge. If W_1 consists of p_1 only, we may omit W_1 and replace W_2 by $(p_1p_2)W_2$, so that G would be decomposable into α paths. Thus W_1 cannot be trivial.

Let W_1' be the path obtained by omitting the first edge p_1p_1' from W_1. As G is a minimal counterexample, the graph $H = G \setminus p_1$ satisfies the assertion.

Now $\{W_1', W_2, \ldots, W_{\alpha+1}\}$ is a dissection of H, so that we can find a dissection $\{Z_1, \ldots, Z_k\}$ of H into $k \leq \alpha$ paths such that the start vertices of these paths are contained in $\{p_1', p_2, \ldots, p_{\alpha+1}\}$.

If p_1' is the start vertex of one of the paths Z_i, Z_i can be replaced by $(p_1 p_1')Z_i$, which yields a dissection of G into at most α paths. If $k < \alpha$, we may add the trivial path $\{p_1'\}$ to the Z_i. If neither of these two conditions holds, we must have $k = \alpha$, and the start vertices of the Z_i are precisely the vertices $p_2, \ldots, p_{\alpha+1}$. Thus p_2 is the start vertex of some Z_h. Replacing Z_h by $(p_1 p_2)Z_h$ again yields a dissection of G into at most α paths. Therefore G cannot be a counterexample, and the assertion holds in general.

7.5.5 As a tournament is an orientation of a complete graph, the maximal independent sets have only one element in this case. Thus the assertion follows immediately from Exercise 7.5.4.

Let us also give a very easy direct proof (not using Exercise 7.5.4). Choose a directed path of maximal length in G, say $W : v_1 \text{ — } v_2 \text{ — } \ldots \text{ — } v_k$. Suppose that W is not a Hamiltonian path; then there exists a vertex v not on W. As W is maximal, G contains neither an edge vv_1 nor an edge $v_k v$, so that G has to contain the edges $v_1 v$ and vv_k. Hence there must be some index i $(1 < i < k)$ such that G contains the edges $v_i v$ and vv_{i+1}. Then we can replace the edge $v_i v_{i+1}$ in W by these two edges, so that W is not maximal, a contradiction.

7.5.9 Let k be the maximal cardinality of a chain in M. Moreover, let A denote the antichain of the maximal elements of M. Then the maximal cardinality of a chain in $M \setminus A$ is $k - 1$, and the assertion follows by induction.

7.5.10 Let $\mathbf{A} = (A_1, \ldots, A_n)$ be a family of subsets of $\{x_1, \ldots, x_m\}$ satisfying (H′). We define a partial ordering on $M = \{x_1, \ldots, x_m, A_1, \ldots, A_n\}$ by

$$u \prec v \quad \Longleftrightarrow \quad u = x_i,\ v = A_j \ \text{ and } \ x_i \in A_j \ (\text{ for suitable } i, j).$$

Let $\{x_1, \ldots, x_h, A_1, \ldots, A_k\}$ be an antichain of maximal cardinality $s = h + k$. Then $k \leq |A_1 \cup \ldots \cup A_k| \leq m - h$, so that $s = h + k \leq m$. By Dilworth's theorem, $(M, \preceq)$ can be decomposed into s chains, say (after renumbering)

$$\{x_1, A_1\}, \ldots, \{x_i, A_i\},\ \{A_{i+1}\}, \ldots, \{A_n\},\ \{x_{i+1}\}, \ldots, \{x_m\}.$$

Then $s = m + n - i$, and hence $n = s - m + i \leq i$; this forces $n = i$, so that $\{x_1, \ldots, x_n\}$ is a transversal of $\mathbf{A}$.

7.7.2 Use Theorem 7.7.3.

7.7.5 We have derived Theorem 7.7.1 from Theorem 6.1.6 by constructing an appropriate flow network N. If c, a, and d are integral, the capacity function of N is likewise integral. Thus Theorem 6.1.5 implies that there exists an integral solution (provided that there are feasible flows).

B.8 Solutions for Chapter 8

8.1.2 Note that each vertex has to have degree at least k if G is k-connected.

8.1.3 Add a new vertex t and all edges xt with $x \in T$ to G. It is easy to show that the resulting graph H is again k-connected: clearly, there is no vertex separator for H consisting of $k-1$ vertices. By Theorem 8.1.1, there are k vertex disjoint paths from s to t; these paths have to contain all the k edges xt with $x \in T$. Deleting these edges, we obtain the desired paths in G.

8.1.6 The graph $K_{m,m+1}$ has connectivity $\kappa = m$ and independence number $\alpha = m+1$. It cannot be Hamiltonian, since a Hamiltonian cycle would have length $2m+1$; by Theorem 3.3.5, bipartite graphs do not contain cycles of odd length.

8.1.7 Using the procedure BLOCK01FLOW of Lemma 6.5.2, we can determine a maximal 0-1-flow as follows (by analogy with Algorithm 6.3.17). Here G is a digraph with two special vertices s and t, and *val* denotes the value of a maximal flow.

Procedure MAX01FLOW$(G, s, t; f, \mathrm{val})$

(1) **for** $e \in E$ **do** $c(e) \leftarrow 1$; $f(e) \leftarrow 0$ **od**
(2) val $\leftarrow 0$; $N \leftarrow (G, c, s, t)$;
(3) **repeat**
(4) AUXNET $(N, f; N'', \max, d)$;
(5) **if** max = false **then** BLOCK01FLOW$(N''; g)$; AUGMENT $(f, g; f)$ **fi**
(6) **until** max = true;
(7) **for** $e \in A_s$ **do**
(8) **if** $f(e) = 1$ **then** val $\leftarrow$ val $+1$ **fi**
(9) **od**

The proofs of Theorems 7.1.1 and 7.1.4 imply that the maximal number of vertex disjoint paths from s to t in G equals the maximal value of a 0-1-flow on the 0-1-network with underlying digraph H defined during the following procedure.

Procedure PATHNR$(G, s, t; k)$

(1) $V' \leftarrow \{s, t\}$; $E' \leftarrow \emptyset$;
(2) **for** $v \in V \setminus \{s, t\}$ **do** $V' \leftarrow V' \cup \{v', v''\}$; $E' \leftarrow E' \cup \{v'v''\}$ **od**
(3) **for** $e \in E$ **do**
(4) **if** $e = sv$ with $v \neq t$ **then** $E' \leftarrow E' \cup \{sv'\}$ **fi**
(5) **if** $e = tv$ with $v \neq s$ **then** $E' \leftarrow E' \cup \{v''t\}$ **fi**
(6) **if** $e = uv$ with $u, v \neq s, t$ **then** $E' \leftarrow E' \cup \{u''v', v''u'\}$ **fi**
(7) **od**
(8) $H \leftarrow (V', E')$; MAX01FLOW $(Hs, t; f, \mathrm{val})$;
(9) **if** $st \in E$ **then** $k \leftarrow \mathrm{val} + 1$ **else** $k \leftarrow$ val **fi**

Theorems 7.1.1 and 7.1.4 show that this procedure is correct; note that s and t are not adjacent in H. If s and t should be adjacent in G, we have to add one further path from s to t, namely the edge st itself. By Corollary 7.1.5, PATHNR has complexity $O(|V|^{1/2}|E|)$. Finally, if G is an undirected graph, we can replace G by its complete orientation (as in the proof of Theorem 7.1.1).

8.2.6 Define a graph G which has a vertex for each junction of the maze, where also the entrance, the exit, and dead ends are viewed as junctions. The edges of G correspond to those paths in the maze which connect two consecutive junctions: the end vertices of an edge are the respective junctions. Figures B.28 and B.29 show the graph G which corresponds to the maze given in Figure 8.3. The labels of the vertices in Figure B.29 indicate one possible course for a DFS on G which starts at the entrance of the maze (which is represented by the vertex labelled 1); the algorithm terminates when the exit is reached (that is, at the vertex labelled 64). The corresponding path through the maze is drawn in Figure B.30; for the sake of simplicity, we have not included dead ends occurring during the DFS (which have, of course, to be traversed and then necessitate corresponding backtracking).

Of course, when we designed the above solution, we had a bird's-eye view of the maze (and used this knowledge). However, it is not hard to find a rule which allows us to apply a DFS to a maze without knowing it in its entirety, provided that it is possible to label junctions and paths when we pass them. We leave it to the reader to formulate such a rule.[4]

8.3.2 Consider two vertices u and v for which $d(u, v)$ is maximal. If v were a cut point, then $G \setminus v$ would consist of two components, so that we could choose a vertex w which is not contained in the component of u. Then every path from u to w would have to contain v, so that the distance from w to u would have to be at least $d(u, v) + 1$, a contradiction. Therefore v and u

[4] In this context, the following quotation from Umberto Eco's *The Name of the Rose* is of some interest; see [Eco83, p. 176]:

> At every new junction, never seen before, the path we have taken will be marked with three signs. If, because of previous signs on some of the paths of the junction, you see that the junction has already been visited, you will make only one mark on the path you have taken. If all the apertures of the junction are still without signs, you will choose any one, making two signs on it. Proceeding through an aperture that bears only one sign, you will make two more, so that now the aperture bears three. All the parts of the labyrinth must have been visited if, arriving at a junction, you never take a passage with three signs, unless none of the other passages is now without signs.

This somewhat chaotic rule contains the basic idea of a depth first search, even though the hero of the tale, William of Baskerville (who admits that he just recites 'an ancient text I once read'), obviously confused the labelling rules a bit.

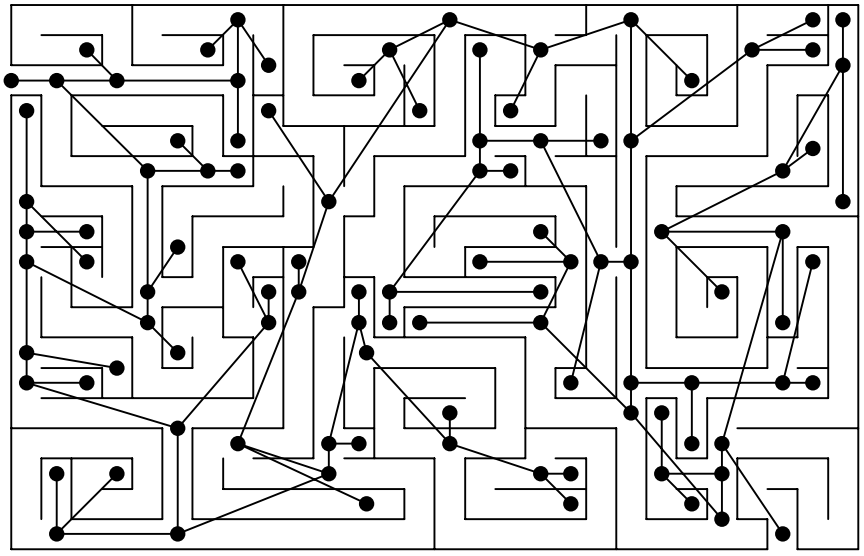


Fig. B.28. A maze with corresponding graph G

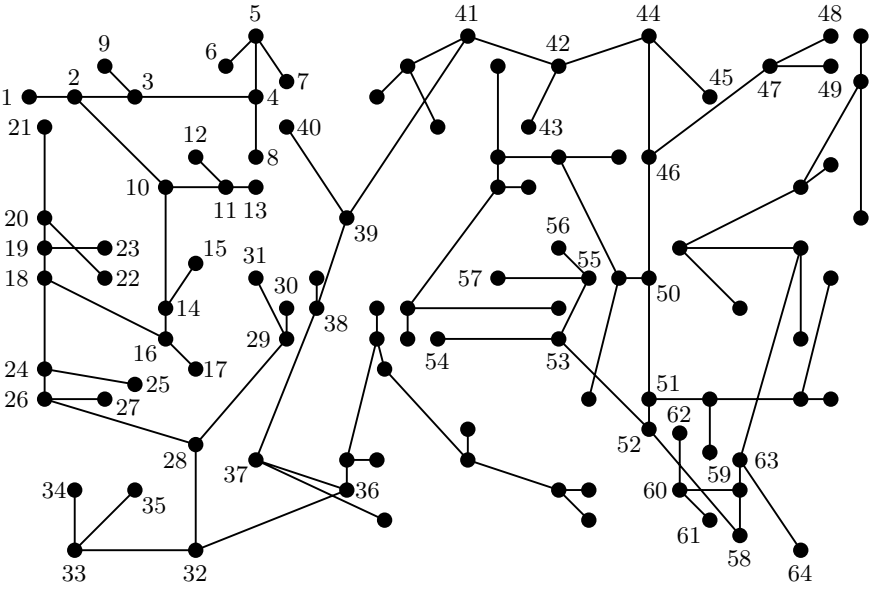


Fig. B.29. A partial DFS on G

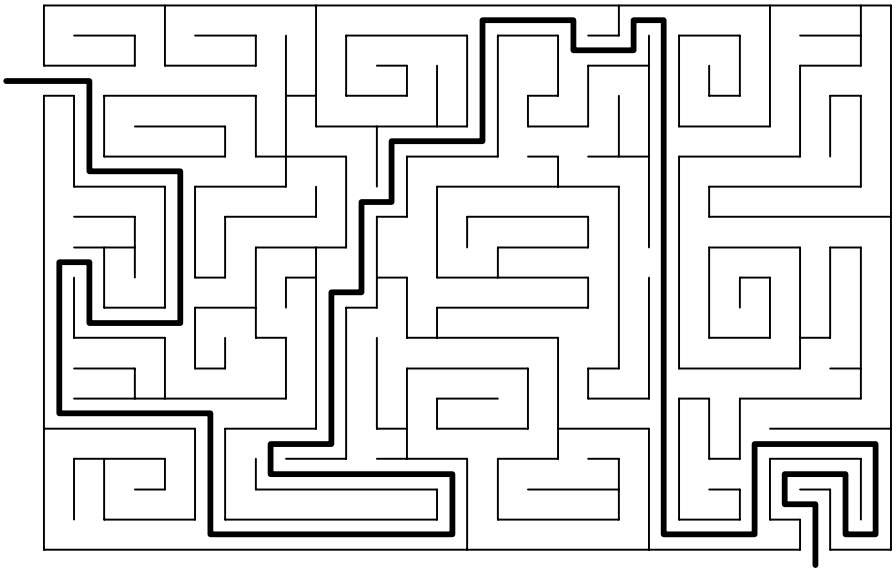


Fig. B.30. A path through the maze

cannot be cut points. On the other hand, a path of length n contains precisely $n-2$ cut points.

8.3.3 Suppose that $bc(G)$ contains a cycle $(B_1, c_1, B_2, c_2, \ldots, B_k, c_k, B_1)$. Then we can remove c_k and still reach vertices in B_1 from vertices in B_k, a contradiction. This proves that $bc(G)$ is always acyclic. If G is connected, also $bc(G)$ is connected, so that $bc(G)$ is a tree. This proves (a).

For claim (b), we may assume that G is connected, so that $p = 1$. Then $bc(G)$ is a tree and, hence, contains precisely $b(G)+c(G)-1$ edges. Each edge connects a cut point with a block, so that the number of edges equals the sum of all the $b(c)$ (over all cut points c). Therefore

$$b(G)+c(G)-1 \;=\; \sum_c b(c) \;=\; \sum_c 1+\sum_v (b(v)-1) \;=\; c(G)+\sum_v (b(v)-1),$$

since each vertex which is not a cut point is contained in precisely one block. Assertion (c) can be proved in a similar manner.

For (d), we use induction on the number $c(G)$ of cut points. The case $c(G) = 1$ is clear. Now assume $c(G) > 1$. Then $bc(G)$ contains a leaf, and every leaf B has to be a block; note that the unique edge incident with B has a cut point c as its other end vertex. Removing B from the graph G corresponds to removing c and B from $bc(G)$. Now the assertion follows by induction.

8.3.4 Let $b(G) = k$. We denote the cardinalities of the blocks by $n_1, \ldots, n_k$ and the number of vertices of G by n. By Exercise 8.3.3 (b), $n_1 + \ldots + n_k = k + n - 1$. By Exercise 8.3.3 (d), a graph with r cut points has to have at least $r+1$ blocks; also, G will have the maximum possible number of edges if and only if each block is a complete graph on at least two vertices. Thus this number is given by

$$\max\left\{\sum_{i=1}^{k}\binom{n_i}{2} : n_1 + \ldots + n_k = n + k - 1; n_1, \ldots, n_k \geq 2; k \geq r+1\right\}$$
$$= \max\left\{k - 1 + \binom{n + k - 1 - (2k-2)}{2} : k \geq r+1\right\} = \binom{n-r}{2} + r,$$

which is realized by a graph consisting of K_{n-r} with a path of length r appended.

8.3.10 We obtain the graph shown in Figure B.31, where each vertex v is labelled with its DFS-number $nr(v)$ and with $L(v)$. Algorithm 8.3.8 yields the cut points i, e, s, and h in this order. The blocks are $\{k, j, i\}$; $\{i, e\}$; $\{e, f, b, a, s\}$; $\{l, h\}$; and $\{h, d, g, c, s\}$. The fat edges are the edges of the DFS tree, and cut points are indicated by a circle.

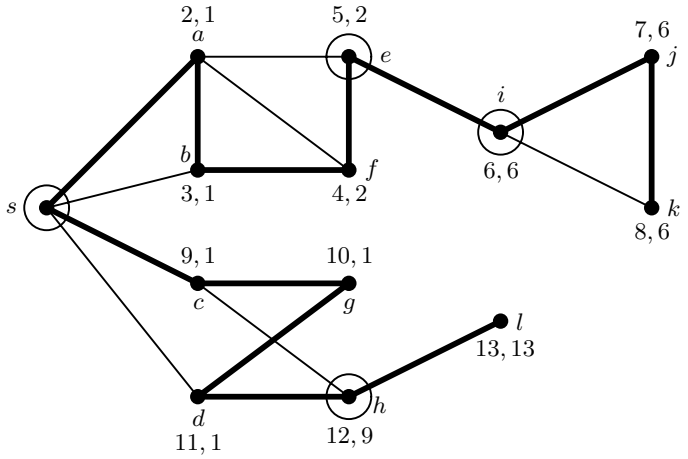


Fig. B.31. DFS-tree, blocks, and cut points

8.4.3 As u is reached later than v during the DFS, the examination of u has to take place during the examination of v.

8.4.4 If a back edge $e = vu$ occurs during the DFS, we obtain a directed cycle in G, as u is an ancestor of v. Conversely, suppose that G contains a

directed cycle. Let v be the first vertex of G examined during the DFS which is contained in a directed cycle, and let $e = uv$ be an edge on such a cycle C. By our choice of v, u is examined later than v during the DFS, so that e is neither a forward edge nor a tree edge. As u is accessible from v (using C), u has to be a descendant of v. Thus e cannot be a cross edge either, so that e must be a back edge.

8.5.2 Choose G to be a directed cycle or the complete orientation of a path.

8.5.3 Let C and C' be two distinct strong components of G. As G is connected, there exists an edge e connecting a vertex in C and a vertex in C'. Then e cannot be contained in a directed cycle, because that would imply $C = C'$. Thus G has to be strongly connected provided that every edge of G is contained in a directed cycle. The converse holds by Theorem 8.5.1.

8.5.8 The vertices h, f, and g each form a strong component with only one element; the remaining vertices together form a further strong component.

8.5.9 Suppose the strong components $C_1, \ldots, C_m$ are contained in a cycle of G'. Then there are edges $v_i v_i'$ with $v_i \in C_i$ and $v_i' \in C_{i+1}$ (where $m + 1$ is interpreted as 1). As C_i contains a directed path from v_{i-1}' to v_i, we obtain a directed cycle, so that $C_1, \ldots, C_m$ have to be contained in a common strong component, a contradiction. Therefore G' has to be acyclic. Figure B.32 shows G' for the digraph G of Figure 3.3.

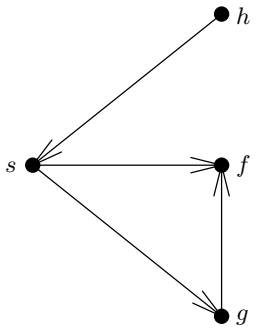


Fig. B.32. Condensed digraph for the digraph of Fig. 3.3

8.5.10 Define a digraph to be strongly k-connected if it is the complete orientation of K_{k+1}, or if each set S of vertices for which $G \setminus S$ is not strongly connected contains at least k vertices. Then the analogues of Theorems 8.1.1 and 8.1.9 hold; in both cases, $\kappa(v_i, w)$ as well as $\kappa(w, v_i)$ have to be calculated.

8.6.2 For $k = m = d$, we can choose $G = K_{d+1}$. For $k \neq d$, we use two copies of the complete graph K_{d+1} on two disjoint vertex sets S and T, together with $2k$ further vertices $x_1, \ldots, x_k, x'_1, \ldots, x'_k$ and all the edges $x_i x'_i$. Moreover, we connect each of the x_i to $d-1$ vertices in S, and each of the x'_i to $d-1$ vertices in T. Finally, we add $m-k$ further edges connecting the vertices in $S \cup \{x_2, \ldots, x_k\}$ to some of the x'_i. See Figure B.33.

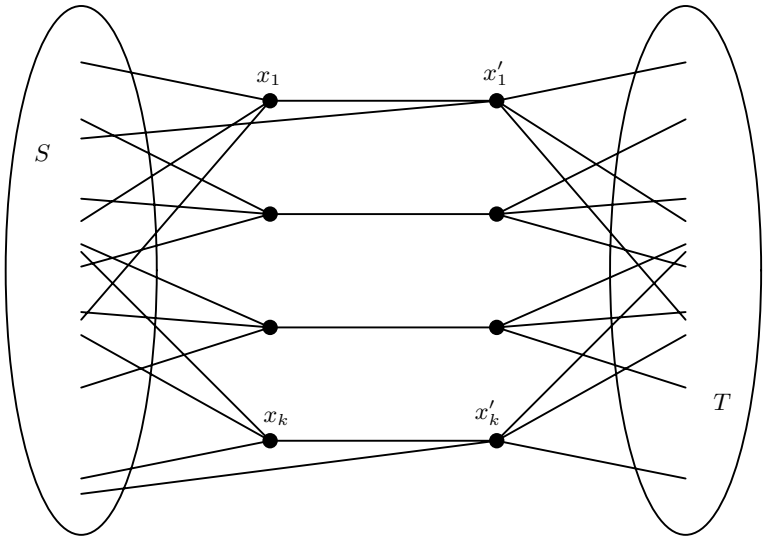


Fig. B.33. Solution to Exercise 8.6.2

8.6.3 Let E' be a minimal edge separator of G. Then $G \setminus E'$ has two connected components S and T, and E' is the cocycle determined by the cut (S, T). We may assume $x = |S| \leq n/2$. Then E' has to contain at least $x\delta - x(x-1) = x(\delta - x + 1)$ edges. It is easy to check $x(\delta - x + 1) \geq \delta$ if $\delta \geq n/2$ (for $x = 1, \ldots, n/2$). The graph consisting of two disjoint copies of K_d (connected by at most $d-1$ edges) shows that nothing can be said for the case $\delta < n/2$.

B.9 Solutions for Chapter 9

9.1.2 If we want to color the icosahedral graph of Figure 9.1, the three vertices of the outer triangle have to get different colors. As any two vertices of this triangle have a further common neighbor, the colors for these three neighbors are now forced (assuming that it is possible to use only three colors); see Figure B.34, where the three colors used are indicated by small gray circles, big gray circles, and big black circles. But now there are vertices of degree 5 for which three neighbors already use up all three colors, so that the coloring

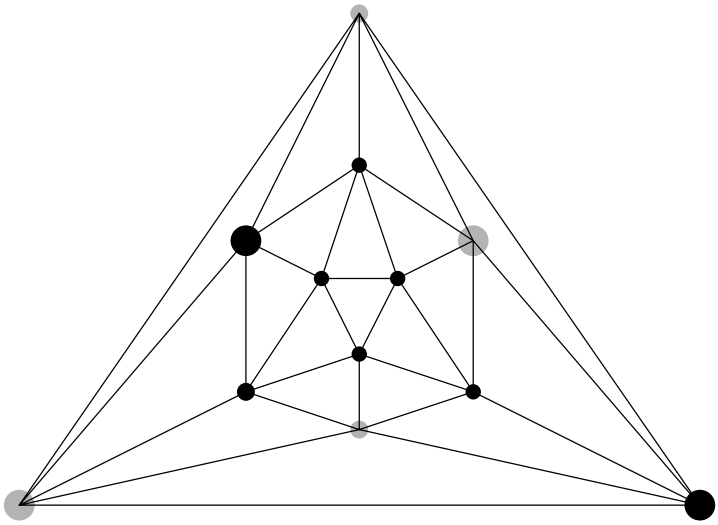


Fig. B.34. A partial 3-coloring of the icosahedral graph

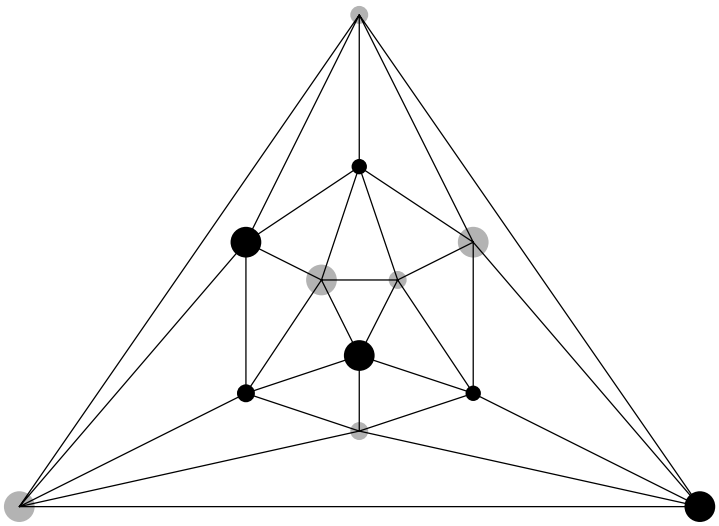


Fig. B.35. A 4-coloring of the icosahedral graph

cannot be completed. If we allow a fourth color, the partial coloring of Figure B.34 can be completed; see Figure B.35.

9.1.9 Note that the Petersen graph is 3-regular; hence $\chi(G) \leq 3$, by Brook's theorem. Now $\chi(G) = 2$ is impossible, as the Petersen graph is not bipartite. Hence $\chi(G) = 3$; see Figure B.36 for an explicit 3-coloring.

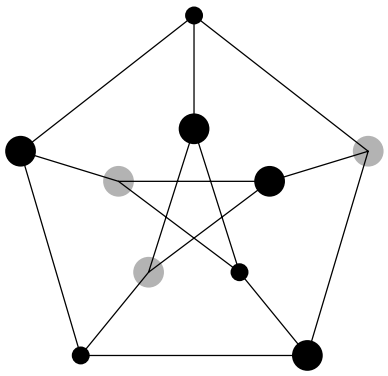


Fig. B.36. A 3-coloring of the Petersen graph

9.2.4 Let $(M_0, M_1, \ldots, M_k = M_0)$ be a sequence of vertices defining a cycle of length $k \geq 4$ in an interval graph, where $M_i = (x_i, y_i)$ for $i = 0, \ldots, k-1$. (The case of closed intervals is similar.) We may assume $x_0 < x_1$. If M_2M_0 is an edge, we have found a chord of the cycle. Otherwise, $M_2 \cap M_0 = \emptyset$, $M_1 \cap M_0 \neq \emptyset$, and $M_2 \cap M_1 \neq \emptyset$ imply $x_1 < y_0 \leq x_2 < y_1$. Thus, if the cycle does not have a chord, the lower bounds of the intervals M_i have to form a monotonically increasing sequence. But then $M_{k-1}M_0$ cannot be an edge, a contradiction. Hence G must be chordal.

9.2.9 As induced subgraphs of a bipartite graph are likewise bipartite, it suffices to prove $\alpha(G) = \theta(G)$ for every bipartite graph G. In the bipartite case, $\theta(G) = |V| - \alpha'(G)$, where $\alpha'(G)$ denotes the maximal cardinality of a matching. Moreover, $\alpha(G) = |V| - \beta(G)$; see Lemma 7.5.1. Therefore Theorem 7.2.3 yields $\alpha(G) = \theta(G)$. (Verifying $\chi(G) = \omega(G)$ is even easier: both parameters are 2 in the bipartite case; see Example 9.1.1.)

9.3.1 Clearly, $\chi'(G)$ is the minimal number of matchings into which G can be decomposed. Thus the assertion amounts to showing that a bipartite graph of maximal degree r can always be decomposed into r matchings, which was proved in the solution to Exercise 7.4.16.

9.3.4 By Exercise 7.2.8, the Petersen graph does not admit a 1-factorization, and hence Corollary 10.3.3 implies $\chi'(G) = 4$. An explicit 4-coloring may be

obtained as follows: take the broken edges in Figure B.27 as one color class. As noted in the solution to Exercise 7.2.8, the complement of M is the union of two vertex disjoint 5-cycles. Trivially, we may color the edges of these two cycles using three further colors.

9.4.6 Clearly, $G = G(H, S)$ is regular of degree k, where k is the cardinality of S. Given any two elements x and y of H, we have to determine the number of elements $z \in H$ which are adjacent to both x and y. As H acts regularly on G, we may assume $y = 1$. By definition, z is adjacent to both 1 and x if and only if $z^{-1}, xz^{-1} \in S$. If we put $d = z^{-1}$ and $c = xz^{-1}$, we may use (9.1) to re-write the preceding condition as

$$x = cd^{-1}, \quad z = d^{-1} \quad \text{with} \quad c, d \in S.$$

Hence the number of elements z of H which are adjacent to both 1 and x equals the number of *quotient representations* of x from S. Noting that x is adjacent to 1 if and only if $x \in S$, one sees that condition (2) in the assertion holds if and only if G is strongly regular with parameters λ and μ.

9.5.3 Denote the two parts of the bipartition of $K_{3,3}$ by S and T, and consider an arbitrary choice of color lists of cardinality ≥ 3 each. If two vertices in the same part, say in S, admit the same color c, we may color them with c, and color the third vertex in S in any admissible way. As this forbids at most two colors and as T is an independent set, we may certainly also color the vertices in the other part T correctly.

Hence we may assume that the color lists for the three vertices in a given part are all disjoint. Thus we have 9 colors available for S, and also for T. Now we just have to pick a color from each of the three lists for S in such a way that the resulting 3-set of colors does not coincide with one of the three color lists for T. Obviously, this is possible (indeed, in many ways).

B.10 Solutions for Chapter 10

10.1.6 Put $b(e) = c(e) = 1$ for each directed edge e of G, and replace each undirected edge $e = \{u, v\}$ by two directed edges $e' = uv$ and $e'' = vu$ with $b(e') = b(e'') = 0$ and $c(e') = c(e'') = 1$; this defines a directed multigraph H with capacity constraints b and c. Obviously, every Euler tour of G yields a feasible circulation on H.

Conversely, let f be a feasible circulation on H. Let e be an undirected edge of G. If either $f(e') = 1$, $f(e'') = 0$ or $f(e'') = 1$, $f(e') = 0$, we replace e by e' or by e'', respectively. Performing this operation for all undirected edges yields a mixed multigraph G' for which the number of directed edges with start vertex v always equals the number of directed edges with end vertex v. Then an Euler tour can be constructed using the methods of Chapter 1; cf. 1.3.1 and 1.6.1.

10.1.7 We introduce the following vertices:

- a source s and a sink t;
- a vertex 0 which represents the person selling the napkins;
- vertices $1, \ldots, N$ corresponding to the dirty napkins which are sent off for cleaning (we assume that all napkins are washed for $i \leq N - n$);
- vertices $1', \ldots, N'$ which represent the supply of clean napkins needed for the N days.

We also add the following edges (with respective capacity constraints):

- $e = s0$ with $b(e) = 0$, $c(e) = \infty$, $\gamma(e) = 0$;
- all si with $b(si) = c(si) = r_i$, $\gamma(si) = 0$;
- all $0i'$ with $b(0i') = 0$, $c(0i') = \infty$, $\gamma(0i') = \alpha$;
- all $e = i(i+m)'$ with $b(e) = 0$, $c(e) = r_i$, $\gamma(e) = \beta$ (for $i + m > N$, the edge $i(i+m)'$ has to be interpreted as it, so that the cost of this edge has to be changed to 0);
- all $e = i(i+n)'$ with $b(e) = 0$, $c(e) = r_i$, $\gamma(e) = \delta$ (for $i + n > N$, the edge $i(i+n)'$ has to be interpreted as it, so that the cost of this edge has to be changed to 0);
- all $i't$ with $b(i't) = c(i't) = r_i$, $\gamma(i't) = 0$;
- all edges $e = i'(i+1)'$ with $b(e) = 0$, $c(e) = \infty$, $\gamma(e) = 0$; these edges represent the possibility of saving unused napkins for the next day.

10.2.3 As before, we define $c'(e) = c(e) - b(e)$. Moreover, put

$$c'(sv) = \sum_{\substack{e^+=v \\ b(e)>0}} b(e) - \sum_{\substack{e^-=v \\ b(e)<0}} b(e); \quad c'(vt) = \sum_{\substack{e^-=v \\ b(e)>0}} b(e) - \sum_{\substack{e^+=v \\ b(e)<0}} b(e).$$

Then Theorem 10.2.1 remains valid without changes: a feasible circulation on G exists if and only if the maximal value of a flow on N is given by $W = \sum_e b(e)$.

10.2.6 First determine – if possible – a feasible flow as in Example 10.2.2. Next, a maximal flow can be found as in Chapter 6. To make sure that this flow is still feasible, we have to replace the condition $f(e) \neq 0$ in step (10) of Algorithm 6.3.14 (when constructing the auxiliary network) by $f(e) > b(e)$ and replace the assignment in step (11) by $c''(e) \leftarrow f(e) - b(e)$. Let us denote the resulting procedure by LEGAUXNET. We may now proceed as in Algorithm 6.3.17. (Note that the algorithm of Ford and Fulkerson with similar changes would serve the same purpose.) We obtain the following algorithm of complexity $O(|V|^3)$, where we put $N = (G, b, c, s, t)$ and use the FIFO preflow push algorithm for determining a blocking flow.

Procedure MAXLEGFLOW(N; legal, f)

(1) Add the edge $r = ts$ to G; $b(r) \leftarrow 0$; $c(r) \leftarrow \infty$;

```
(2) LEGCIRC(G, b, c; f, legal);
(3) if legal = true then
(4)     remove the edge r = ts from G;
(5)     repeat
(6)         LEGAUXNET(N, f; N'', max, d);
(7)         if max = false then BLOCKMKM(N''; g); AUGMENT(f, g; f) fi
(8)     until max = true
(9) fi
```

10.2.9 Apply the criterion of Theorem 10.2.7 to the directed multigraph H defined in the solution to Exercise 10.1.6 (using the capacity functions b and c given there); this yields the following theorem.

Let G be a connected mixed multigraph. Then G has an Euler tour if and only if the following two conditions hold:

(i) *Each vertex of G is incident with an even number of edges.*

(ii) *For each subset X of V, the difference between the number of directed edges e with $e^- \in X$ and $e^+ \in V \setminus X$ and the number of directed edges e with $e^+ \in X$ and $e^- \in V \setminus X$ is at most as large as the number of undirected edges connecting X and $V \setminus X$.*

10.2.10 Similarly to the proof of Theorem 10.2.8, one sees that the minimal value of a feasible flow is given by

$$\max\left\{\sum_{e^-\in S, e^+\in T} b(e) - \sum_{e^+\in S, e^-\in T} c(e)\colon (S,T) \text{ is a cut on } N\right\},$$

where has $c(r) = v$ and $b(r) = -\infty$ for the return arc $r = ts$.

To determine a minimal flow, an arbitrary feasible flow can be changed applying methods similar to those used in Chapter 6. To find a path along which the value of the flow can be decreased, we admit forward edges e in the auxiliary network if and only if $b(e) < f(e)$, and backward edges if and only if $f(e) < c(e)$. Then the bounds on the complexity are the same as in Chapter 6. We leave the details to the reader.

10.2.11 By Exercise 8.5.3, G is strongly connected if and only if every edge is contained in a directed cycle. Hence we may show that this criterion is satisfied if and only if G has a feasible circulation.

First assume that G has a feasible circulation. Let $e = uv$ be an edge of G, and let S be the set of all vertices s from which u is accessible. If $v \notin S$, then $(S, V \setminus S)$ is a cut for which all edges of the corresponding cocycle are oriented from S to $V \setminus S$. Such a cut would violate the condition of Theorem 10.2.7, as $b(e) > 0$. Therefore there exists a directed path W from v to u. Then e is contained in the directed cycle $u \overset{e}{\text{---}} v \overset{W}{\text{---}} u$.

Conversely, assume that every edge of G is contained in a directed cycle. Then all cocycles contain edges in both possible directions, so that the condition of Theorem 10.2.7 is satisfied, since $c(e) = \infty$ for all e.

Finally, let N be a flow network with $c(e) = \infty$ and $b(e) > 0$ for all edges e. Removing the return arc $r = ts$, we see that a feasible flow exists if and only if each edge is contained either in a directed cycle or in a directed path from s to t.

10.3.4 The first assertion is an immediate consequence of condition (Z1):

$$\begin{aligned} f(S,T) &= \sum_{e^- \in S, e^+ \in T} f(e) = \sum_{v \in S} \sum_{e^- = v, e^+ \in T} f(e) \\ &= \sum_{v \in S} \Big(\sum_{e^- = v} f(e) - \sum_{e^- = v, e^+ \in S} f(e) \Big) \\ &= \sum_{v \in S} \Big(\sum_{e^+ = v} f(e) - \sum_{e^- = v, e^+ \in S} f(e) \Big) \\ &= \sum_{v \in S} \Big(\sum_{e^+ = v, e^- \in T} f(e) + \sum_{e^+ = v, e^- \in S} f(e) - \sum_{e^- = v, e^+ \in S} f(e) \Big) \\ &= \sum_{e^- \in T, e^+ \in S} f(e) = f(T,S). \end{aligned}$$

For the second assertion, let $f \neq 0$ be a circulation and consider an edge $e = uv$ in the support of f. Suppose that e is a bridge. Then $G \setminus e$ has at least two connected components S and T, say $u \in S$ and $v \in T$. Now e is the only edge in the cocycle $E(S,T)$, so that $f(S,T) = f(e) \neq 0$ and $f(T,S) = 0$, contradicting the first assertion.

10.3.7 We may assume that each of the elementary circulations f_e in the proof of Theorem 10.3.6 satisfies the condition $f_e(e) = 1$ (otherwise we multiply f_e by -1). Given an arbitrary circulation f, put

$$g = f - \sum_{e \in G \setminus T} f(e) f_e.$$

Then the support of g is contained in T, and Corollary 10.3.3 yields $g = 0$.

10.3.8 Denote the vector in $\mathbb{R}^m$ corresponding to $\delta q : E \to \mathbb{R}$ by $\delta \mathbf{q}$. Then

$$\delta \mathbf{q} = \sum_{i=1}^{n} q(v_i) \mathbf{a}_i,$$

where $\mathbf{a}_i$ is the i-th row of M (corresponding to the vertex i). Now P corresponds to the row space of M, and Theorem 4.2.4 yields $\dim P = \operatorname{rank} M = n - p$. This shows part (a).

For part (b), let (S,T) be a cut of G. We put $q(v) = 1$ for $v \in S$, and $q(v) = 0$ for $v \in T$. Then $\delta q(e) = +1$ or $= -1$ for all edges e contained in the cocycle corresponding to (S,T), and $\delta q(e) = 0$ for all other edges.

Finally, let T be a spanning tree and T' the corresponding cotree: $T' = E \setminus T$. Consider any edge $e \in T$. By Lemma 4.3.2, there exists a unique cut for which the corresponding cocycle C_e contains only edges in T', except for e. By part (b), there is a potential difference δq_e for C_e whose support consists precisely of the edges of C_e. Thus the δq_e with $e \in T$ are $n-1$ linearly independent potential differences; in view of part (a), they have to form a basis of P: as G is connected, $p = 1$.

10.3.16 First assume the existence of a cycle K as described in the first case of the painting lemma. As no edge of K is uncolored, we have $b(e) < c(e)$ for all $e \in K$. Now every black edge of K has the same orientation as e_0 and satisfies $f(e) < c(e)$; every green edge of K has the opposite orientation as e_0 and satisfies $b(e) < c(e)$; and every red edge satisfies both of the previous conditions, that is, $b(e) < f(e) < c(e)$. If we traverse K in the direction given by e_0, black edges are forward edges, while green edges are backward edges; red edges may be either forward or backward edges. The previous remarks show that we may define a new circulation f' by increasing f on all forward edges and decreasing f on all backward edges (by a sufficiently small amount δ), without increasing any of the deviation values $d(e)$. On the contrary, at least one deviation will become strictly smaller: $d(e_0)$ is replaced by $d(e_0) - \delta$. Thus f' is indeed a better circulation than f: it satisfies $D(f') \leq D(f) - \delta$.

Now assume the existence of a cocycle C as described in the second case of the painting lemma. Let (S,T) be the cut defining C; we may assume that e_0 is oriented from T to S. Similarly to the first case, every forward edge of C (that is, every edge with the same orientation as e_0 in C) satisfies $f(e) \leq b(e)$, while every backward edge satisfies $c(e) \leq f(e)$. Moreover, we have strict inequality for the forward edge e_0, namely $f(e_0) < b(e_0)$. Hence, using Exercise 10.3.4,

$$c(S,T) \;\leq\; f(S,T) \;=\; f(T,S) \;<\; b(T,S).$$

On the other hand, the circulation theorem 10.2.7 gives the necessary condition $b(T,S) \leq c(S,T)$ for the existence of a feasible circulation. These inequalities are inconsistent, and hence no feasible circulation can exist in the second case.

Finally, note that the value δ in the first case may always be chosen as an integer, since the capacities are integral by hypothesis. Using induction, the value $D = D(f)$ likewise is integral throughout the entire procedure. As D strictly decreases and is bounded from below by 0, the procedure terminates with either a feasible circulation or a cocycle certifying the nonexistence of such a circulation.

10.3.17 Note that the necessity of the criterion given in the circulation theorem is rather trivial: in view of Exercise 10.3.4, any feasible circulation f

satisfies

$$c(S,T) \;\geq\; f(S,T) \;=\; f(T,S) \;\geq\; b(T,S).$$

Now assume that this criterion is satisfied and that the capacities are integral. Then the second case in the algorithm of Herz cannot occur (as shown in the solution of Exercise 10.3.16), and hence the algorithm terminates with a feasible circulation.

10.4.4 For any feasible circulation, the integer

$$M \;=\; \sum_{\substack{e \\ \gamma(e) > 0}} \gamma(e)c(e) + \sum_{\substack{e \\ \gamma(e) < 0}} \gamma(e)b(e)$$

is an upper bound for the cost. Defining m as in the proof of Lemma 10.4.2, $M - m$ is an upper bound for the number of iterations needed.

10.5.4 We first consider the problem of determining the optimal cost $\gamma(v)$, where M is the maximal value of a flow on N and $v \leq M$ a real number. Denote the largest integer $\leq v$ by w, and let f be an optimal flow of value w constructed by Algorithm 10.5.2. Moreover, let W be an augmenting path of least possible cost from s to t in the auxiliary network $N'(f)$ with respect to the cost function γ'. As W has integral capacity, f can be augmented along W by $\delta = v - w < 1$ with cost $\delta\gamma'(W)$. It can be shown that the resulting flow of value v is optimal (proceed as in the proof of Lemma 10.5.1).

Thus the cost function is linear between any two integers w and $w+1$. As the cost of an augmenting path is always nonnegative, the cost function is also monotonically increasing. Finally, for any two feasible flows f and f' of values v and v', respectively, and for each λ with $0 \leq \lambda \leq 1$, the linear combination $\lambda f + (1-\lambda)f'$ is a feasible flow with value $\lambda v + (1-\lambda)v'$, so that

$$\gamma(\lambda v + (1-\lambda)v') \;\leq\; \lambda\gamma(v) + (1-\lambda)\gamma(v').$$

Hence the cost function is a monotonically increasing, piecewise linear, convex function.

10.5.5 By Example 10.1.4, the assignment problem can be reduced to the determination of an optimal flow of value n on a flow network with $2n+2$ vertices. As all capacities are integral (actually, they are always 1) and as the cost function is nonnegative, the algorithm of Busacker and Gowen can be used for determining an optimal flow with complexity $O(|V|^2 n) = O(n^3)$, by Theorem 10.5.3. Hence the assignment problem has complexity at most $O(n^3)$; it will be studied more thoroughly in Chapter 14.

10.6.2 The following procedure provides a possible solution:

Procedure RESIDUAL$(G, c, f; H)$

(1) $E' \leftarrow \emptyset$;
(2) **for** $e \in E$ **do**
(3) **if** $c(e) > f(e)$ **then** $E' \leftarrow E' \cup \{e\}$ **fi**
(4) **od**
(5) $H \leftarrow (V, E')$

10.6.8 Let f be an ε-optimal pseudoflow on (G, c) with respect to the cost function γ. We construct the auxiliary graph H_f described in the proof of Theorem 10.6.6 with cost function $\gamma^{(\varepsilon)}$, and proceed by determining an SP-tree for $(H_f, \gamma^{(\varepsilon)})$ using the procedure SPTREE given in Exercise 3.10.3; then the desired potential is just the distance function in this network, by Corollary 10.6.7. The procedure below does the job; here s is a vertex not contained in G.

Procedure POTENTIAL$(G, c, \gamma, f, \varepsilon; p)$

(1) RESIDUAL$(G, c, f; H)$;
(2) $V^* \leftarrow V \cup \{s\}$; $E^* \leftarrow E'$;
(3) **for** $e \in E$ **do** $\gamma^*(e) \leftarrow \gamma^*(e) + \varepsilon$ **od**
(4) **for** $v \in V$ **do** $E^* \leftarrow E^* \cup \{sv\}$; $\gamma^*(sv) \leftarrow 0$ **od**
(5) $H^* \leftarrow (V^*, E^*)$;
(6) SPTREE $(H^*, \gamma^*, s; p, q, \text{neg}, T)$

Note that p is the required distance function d_T in the arborescence T; the remaining output variables (that is, the predecessor function q for the SP-tree T and the Boolean variable neg) are not actually needed here. We could, of course, use the condition neg = false to check whether the given pseudoflow is indeed ε-optimal; see Theorem 10.6.6.

10.6.13 In the following procedure, s is a vertex not contained in H, and $n - 1$ denotes the number of vertices of H.

Procedure MEANCYCLE$(H, w; \mu, C)$

(1) TOPSORT $(H; \text{topnr}, \text{acyclic})$;
(2) **if** acyclic = true
(3) **then** $\mu \leftarrow \infty$
(4) **else** $V^* \leftarrow V \cup \{s\}$; $E^* \leftarrow E$; $F(0, s) \leftarrow 0$;
(5) **for** $v \in V$ **do**
(6) $E^* \leftarrow E^* \cup \{sv\}$;
(7) $w(sv) \leftarrow 0$; $F(0, v) \leftarrow \infty$
(8) **od**
(9) **for** $k = 1$ **to** n **do**
(10) **for** $v \in V^*$ **do**
(11) $F(k, v) \leftarrow \min\{F(k-1, u) + w(uv) : uv \in E^*\}$
(12) $q(k, v) \leftarrow u$, where $u \in V$ is an element such that
$F(k-1, u) + w(uv) = \min\{F(k-1, x) + w(xv) : xv \in E^*\}$;

(13) **od**
(14) **od**
(15) **for** $v \in V$ **do**
(16) $M(v) \leftarrow \max\left\{\frac{F(n,v)-F(k,v)}{n-k} : k = 0, \ldots, n-1\right\}$
(17) **od**
(18) choose v with $M(v) = \min\{M(x) : x \in V\}$;
(19) $\mu \leftarrow M(v)$;
(20) determine a walk W of length $F(n, v)$ from s to v which consists of n edges;
(21) determine a cycle C contained in W
(22) **fi**

To prove that this procedure is correct, we use the proofs of Theorem 10.6.11 and Corollary 10.6.12. The procedure TOPSORT checks – according to Theorem 2.6.6 – whether H^* (and hence H) is acyclic; in this case, μ is set to ∞. Otherwise, H contains directed cycles, and the **for**-loop in steps (9) to (14) determines the minimal length $F(k, v)$ of a directed walk from s to v consisting of precisely k edges (for all k and v); this is done recursively. Then, in steps (15) to (19), the minimum cycle mean μ of a directed cycle in H is calculated in accordance with Theorem 10.6.11. Now consider – as in the proof of Theorem 10.6.11 – the changed weight function w' defined by $w'(e) = w(e) - \mu$ for all $e \in E$. The second part of the proof of Theorem 10.6.11 shows that the corresponding values $F'(k, v)$ and the vertex v chosen in step (18) satisfy the condition

$$\max\left\{\frac{F'(n,v) - F'(k,v)}{n-k} :\ k = 0, \ldots, n-1\right\} \ =\ 0.$$

Thus the network (H, w') has minimum cycle mean 0. Now the first part of the proof of Theorem 10.6.11 shows that $F'(n, v) = F(n, v) - n\mu$ is the shortest length of a directed walk from s to v (and therefore the distance from s to v) in (H^*, w'). In step (20), a directed walk W from s to v having this length and consisting of n edges is determined; this is done recursively using the function $q(k, v)$ defined in step (12): the last edge of W is uv, where $u = q(n, v)$; the edge before the last is $u'u$, where $u' = q(n, u)$ and so on.

As W consists of precisely n edges, W has to contain a directed cycle C which is determined in step (21): this can be implemented, for example, by a labelling process while W is traced from s to v. Then $W \setminus C$ is a directed walk from s to v as well, which must have length at least $F'(u, v)$ in (H^*, w'). Therefore $w'(C)$ has to be 0; otherwise, $w'(C)$ would be positive because of $\mu' = 0$, so that $w'(W \setminus C) < w'(W)$. Hence $w(C) = \mu$.

10.6.15 Using Exercise 10.6.13 and Theorem 10.6.14, we obtain the following procedure:

Procedure TIGHT $(G, c, \gamma, f; \varepsilon)$

(1) RESIDUAL$(G, c, f; H)$;
(2) MEANCYCLE$(H, \gamma; \mu, C)$;
(3) **if** $\mu \geq 0$ **then** $\varepsilon \leftarrow 0$ **else** $\varepsilon \leftarrow -\mu$ **fi**

10.8.9 Define the function Φ as given in the hint. At the beginning of Algorithm 10.8.1, $\Phi \leq |V|$, since the admissible graph G_A does not contain any edges at this point, so that $\Phi(v) = 1$ holds trivially for all vertices.

A saturating PUSH-operation, say PUSH(u, v), can increase Φ by at most $\Phi(v) \leq |V|$ (if v becomes active by this operation), so that all the saturating PUSH-operations together can increase Φ by at most $O(|V|^2|E|)$, by Lemma 10.8.7. A RELABEL(v)-operation might add new edges of the form vu to G_A, so that Φ is increased by at most $|V|$. Note that RELABEL(v) does not change the values $\Phi(w)$ for $w \neq v$: as we saw in the proof of Lemma 10.8.8, G_A does not contain any edges with end vertex v after this operation. By Lemma 10.8.6, all the RELABEL-operations together can increase Φ by at most $O(|V|^3)$; this value is dominated by $O(|V|^2|E|)$.

It remains to consider the non-saturating PUSH-operations. Such a PUSH(u, v) makes u inactive, whereas v might become active; thus it decreases Φ by $\Phi(u)$, and possibly increases Φ by $\Phi(v)$. However, $\Phi(u) \geq \Phi(v) + 1$, since each vertex in G_A which is accessible from v is accessible from u as well, and since u is not accessible from v (as G_A is acyclic by Lemma 10.8.8). Note that a PUSH-operation does not add any edges to G_A according to the proof of Lemma 10.8.8. Thus each non-saturating PUSH decreases Φ by at least 1. It follows that the total number of non-saturating PUSH-operations is bounded by the total increase of Φ during the algorithm, which is $O(|V|^2|E|)$.

10.9.6 The circulation f constructed during the initialization of Algorithm 10.9.1 is clearly C-optimal, so that $\varepsilon(f_0) \leq C$. By Lemma 10.9.3, $|E|$ consecutive iterations decrease $\varepsilon(f)$ by at least a factor of $1 - 1/|V|$. Theorem 10.6.5 guarantees that the algorithm terminates with an optimal circulation f as soon as $\varepsilon(f)$ becomes smaller than $1/|V|$; hence it suffices to decrease $\varepsilon(f)$ by a total factor of value $< 1/C|V|$. By Theorem 10.6.4, $|E||V|$ consecutive iterations always decrease $\varepsilon(f)$ by at least a factor of $1/2$, so that the algorithm has to terminate with an optimal circulation after at most $O(|V||E| \log C|V|)$ iterations.

10.11.2 Axioms (MS1) and (MS2) for a metric space hold trivially. We need to check the triangle inequality (MS3). Let $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ be three words in S^n. Denote by X the set of indices for which $\mathbf{x}$ and $\mathbf{y}$ disagree, so that $d(\mathbf{x}, \mathbf{y}) = |X|$; similarly, let Z be the set of indices for which $\mathbf{z}$ and $\mathbf{y}$ disagree. Then all three words agree for all indices not in $X \cup Z$, and hence

$$d(\mathbf{x}, \mathbf{z}) \;\leq\; |X \cup Z| \;=\; |X| + |Z| - |X \cap Z| \;\leq\; d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z}).$$

10.11.4 By Exercise 4.2.14, the incidence matrix M of G has rank $n - 1$, also when considered as a binary matrix. Note that a binary vector $\mathbf{f}$ satisfies

$M\mathbf{f} = 0$ over $\mathbb{Z}_2$ if and only if $\mathbf{f}$ is the incidence vector of an even subgraph of G. Hence $C_E(G)$ has dimension $m - \operatorname{rank} A = m - n + 1$, as in the case of circulations. (This again shows that the even subgraphs of G may be viewed as the *binary circulations* on G.)

10.11.17 Let $a \geq 2$. The even graphical code of $K_{p,p'}$ with $p = 2^{a+1}$ and $p' = 2^a$ has parameters $[2^{2a+1}, 2^{2a} - 2^{a+1} - 2^a + 1, 4]$. We apply Theorem 10.11.15 with $c = 2$, $n_1 = 2^{a+1}$, and $n_2 = 2^a$. Then we may use for O_1 the extended binary Hamming code with parameters $[2^{a+1}, 2^{a+1} - (a+1) - 1, 4]$, and for O_2 the extended binary Hamming code with parameters $[2^a, 2^a - a - 1, 4]$. This results in a graphical code C^* with parameters $[2^{2a+1}, 2^{2a+1} - (2a+1) - 1, 4]$. By the remarks preceding Example 10.11.16, C^* is in fact the extended binary Hamming code with these parameters.

10.11.18 We have to show that the extended binary Hamming codes with parameters $[2^h, 2^h - h - 1, 4]$ can be constructed recursively as purely graphical codes. The case $h = 2$ is realized by the even graphical code of a cycle of length 4, which indeed has parameters $[4, 1, 4]$. The case $h = 3$ can be obtained from the even graphical code belonging to $K_{4,2}$, an $[8, 3, 4]$ code, using an augmentation according to Lemma 10.11.10. Now we can use induction on h, applying the constructions in Example 10.11.16 (for even values of h) and Exercise 10.11.17 (for odd values of h).

B.11 Solutions for Chapter 11

11.3.3 First assume $xv \in E'$, that is, $d(v) + \sum_{e^+=v} b(e) - \sum_{e^-=v} b(e) < 0$. With $g(e) = b(e)$ for all $e \in E$, the demand restriction for v yields

$$\begin{aligned} d(v) \;=\; d'(v) \;&=\; g(xv) + \sum_{e^+=v} g(e) - \sum_{e^-=v} g(e) \\ &=\; g(xv) + \sum_{e^+=v} b(e) - \sum_{e^-=v} b(e), \end{aligned}$$

and hence

$$g(xv) \;=\; d(v) - \sum_{e^+=v} b(e) + \sum_{e^-=v} b(e) \;=\; c'(xv) - 1.$$

Thus indeed $g(xv) = h(xv)$, where h is the admissible flow defined in the first part of the proof of Theorem 11.3.1. Similarly, one checks $g(vx) = h(vx)$ whenever $vx \in E'$.

B.12 Solutions for Chapter 12

12.1.4 As the proof of Theorem 12.1.1 shows, every maximal spanning tree for (G, w) is also an equivalent flow tree for $N = (G, c)$. Conversely, let T be

an equivalent flow tree for N. Note that the flow value $w_T(x, y)$ between x and y in the network $(T, w|T)$ equals the capacity $w(P_{xy})$ of the unique path P_{xy} from x to y in T. By hypothesis, $w_T(x, y) = w(x, y)$ for all $x, y \in V$, which implies that P_{xy} is a path of maximal capacity from x to y in the network (G, w). By Exercise 4.5.6, T is a maximal spanning tree for (G, w).

12.1.5 We use induction on the number n of vertices. The case $n = 2$ is trivial. Thus let $n \geq 3$. Choose a pair (x, y) of vertices such that $w(x, y)$ is maximal, and remove one of these vertices, say x. By the induction hypothesis, the smaller flow network on $G \setminus x$ can be realized on a path P, say

$$P: \; x_1 \;\text{—}\; x_2 \;\text{—}\; \ldots \;\text{—}\; x_{n-1},$$

where $y = x_i$. We insert x after y in P and denote the resulting path by P'. As $w(x, y)$ is the largest flow value on N, the flow values realized before on $G \setminus x$ are not changed by this operation. Clearly, we also obtain the correct flow value $w(x, y)$ between x and y.

It remains to consider $w(x, z)$ for a vertex z with $z \neq x, y$. Then $w(x, z) = w(y, z)$: the inequality (12.1) of Theorem 12.1.1 shows

$$w(x, z) \;\geq\; \min\{w(x, y), w(y, z)\} \;=\; w(y, z);$$

similarly, $w(y, z) \geq w(x, z)$. As P realizes all flow values $w(y, z)$ correctly, P' yields the correct values $w(x, z)$.

Applying this technique recursively, we obtain from the network of Figure 12.1 the flow networks on smaller trees shown in Figure B.37 (in the order shown there). These smaller flow networks can be realized (beginning with the trivial path on two vertices) on the paths shown below the corresponding tree.

12.3.5 For the graph in Example 12.3.1, $u(a) = 13$, $u(b) = 13$, $u(c) = 12$, $u(d) = 12$, $u(h) = 13$, $u(g) = 15$, $u(f) = 15$, and $u(e) = 11$. As shown in the proof of Theorem 12.3.4, the increased flow requirements which can be realized with the minimal capacity of 52 given by r (see Figure 12.11) are $s(x, y) = \min\{u(x), u(y)\}$. Using this weight function on K yields the same dominating tree T as in Example 12.3.1; only the weights differ, see Figure B.38.

Now we decompose T into the uniform trees $U_1, \ldots, U_4$ shown in Figure B.39 and construct corresponding cycles, say $C_1 = (a, b, c, d, e, f, g, h, a)$ with weight 11/2, $C_2 = (a, b, c, d, f, g, h, a)$ with weight 1/2, $C_3 = (a, b, f, g, h, a)$ with weight 1/2, and the edge $C_4 = (g, f)$ with weight 2 (recall the order of the vertices is arbitrary); this yields the dominating network N shown in Figure B.40. Note that N indeed allows higher flow values: for example, $w(a, c) = 12$, whereas the network of Figure 12.11 gives a flow between a and c of value 8 only.

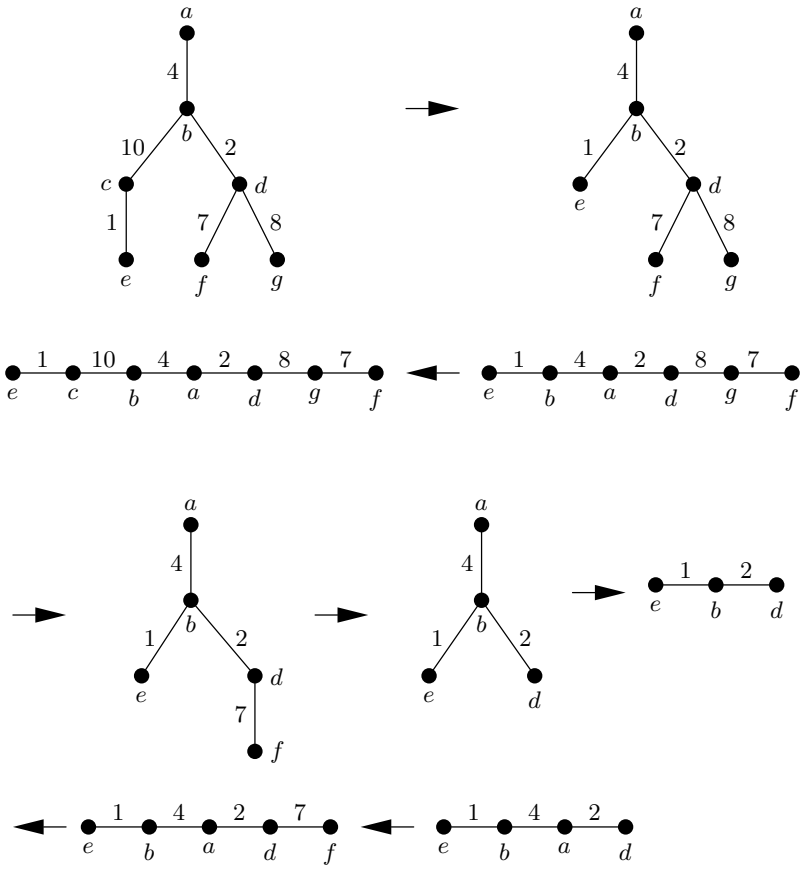


Fig. B.37. Recursive realization of a flow network on a path

12.4.9 The network for the given values of the request function is shown in Figure B.41. By Theorem 12.4.6, we have to determine a cut tree T for (G, r); this is done using Algorithm 12.4.2. After initializing T as a star with center 1, we obtain $s = 2$, $t = 1$, $w = 18$, and $s = \{2, 3, 4, 5, 6\}$, so that $f(s) = 2$. The vertices $3, 4, 5$, and 6 are then cut off from 1 and connected to 2 instead.

Next we have $s = 3$, $t = 2$, $w = 13$, and $S = \{3, 4, 5\}$. We set $f(3) = 13$, cut off the vertices 4 and 5 from 2, and connect them to 3 instead. For $s = 4$, we get $t = 3$, $w = 14$, and $S = \{4\}$. The tree T is not changed during this iteration, we just set $f(4) = 14$.

Next $s = 5$, $t = 3$, $w = 15$, and $S = \{4, 5, 6\}$. The vertices 3, 4, and 5 are removed from T, $s = 5$ is then connected to $p(t) = p(3) = 3$, and 3 and 4 are connected to 5. Also, $f(5)$ is now given the value $f(t) = f(3) = 13$, and $f(3)$ is changed to $w = 15$.

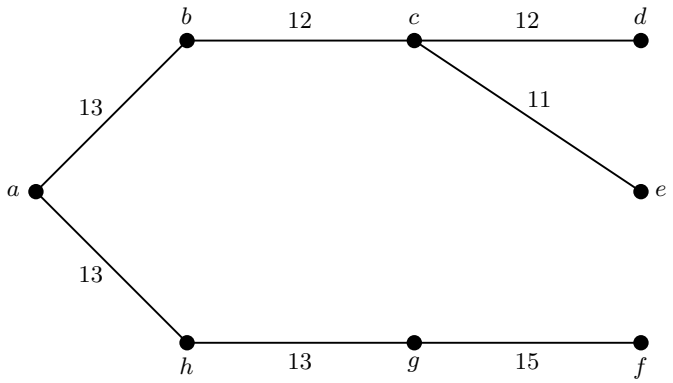


Fig. B.38. Dominating tree T

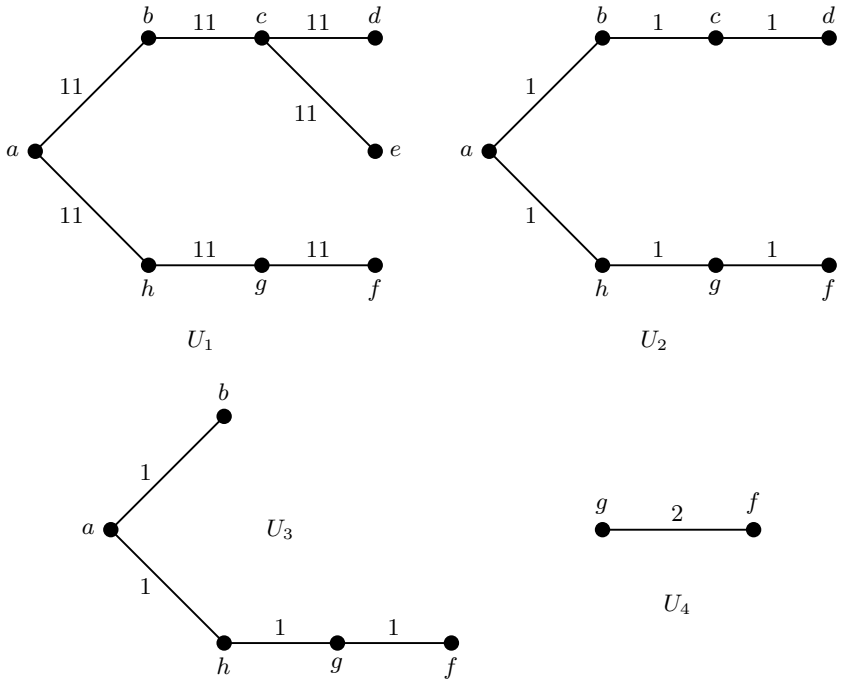


Fig. B.39. Partitioning T into uniform trees

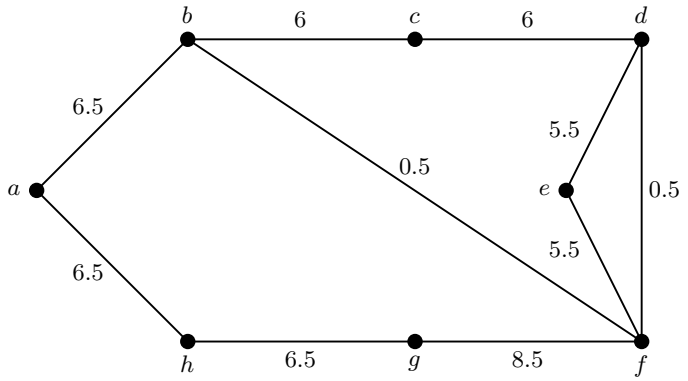


Fig. B.40. A dominating network

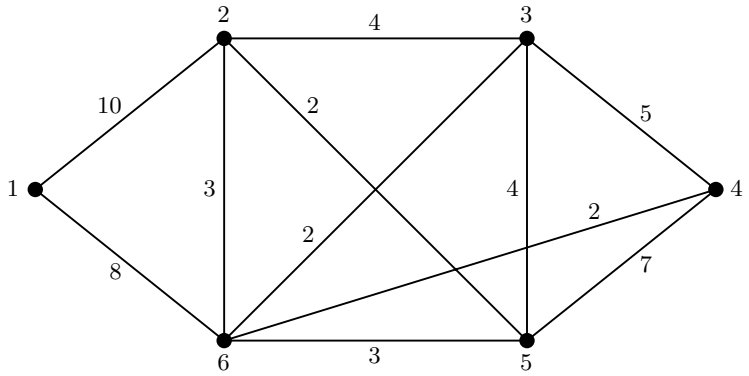


Fig. B.41. Network for Exercise 12.4.9

In the final iteration, $s = 6$, $t = 2$, $w = 17$, and $S = \{3, 4, 5, 6\}$. We set $f(6) = 17$, and cut off 5 from 2 and re-connect it to 6. The resulting tree with weight 77 solves Problem 12.4.4 for the given request function r. Figure B.42 illustrates how the algorithm works.

12.5.2 The relevant part of the auxiliary network corresponding to the flow g of Figure 12.16 is drawn in Figure B.43; the fat edges form an augmenting path with cost 3 and capacity 15. Increasing the capacity of both sb and ct by θ (for $\theta = 1, \ldots, 15$), we obtain a flow of value $v = 41 + \theta$. The total cost for the corresponding increase of the capacity is $20 + 3\theta$. In particular, we obtain the flow h of value 56 and cost 65 shown in Figure B.44.

The next step yields the auxiliary network shown in Figure B.45; again, the fat edges form an augmenting path, now with cost 4 and unlimited capacity. Thus we can now realize any flow value $v = 56 + \tau$ with total cost $65 + 4\tau$

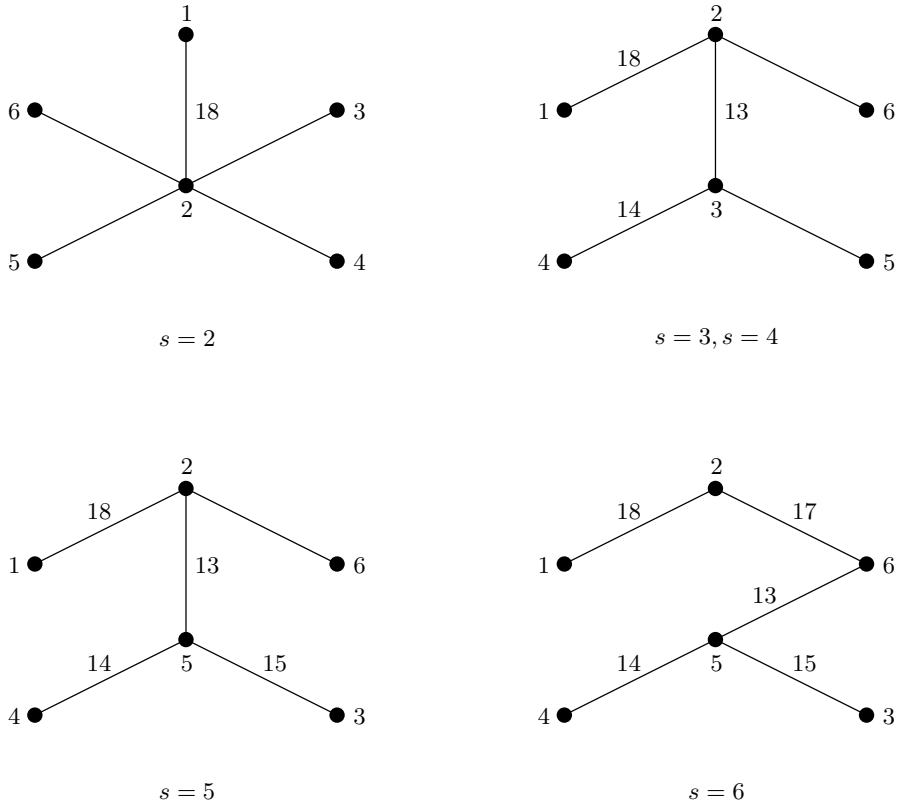


Fig. B.42. Determining a cut tree for N

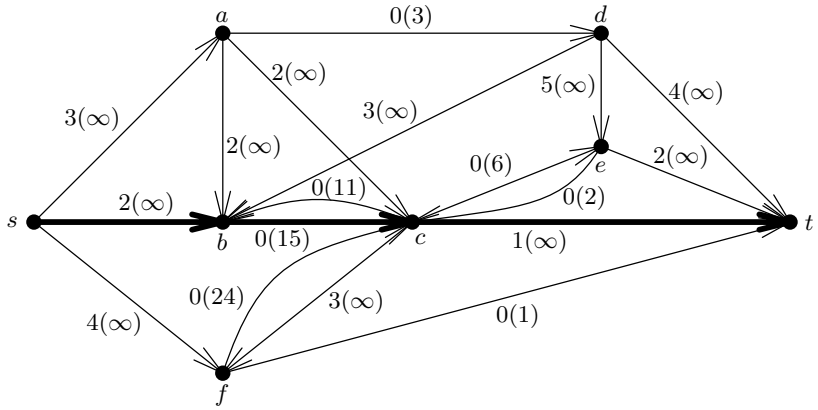


Fig. B.43. Auxiliary network for g

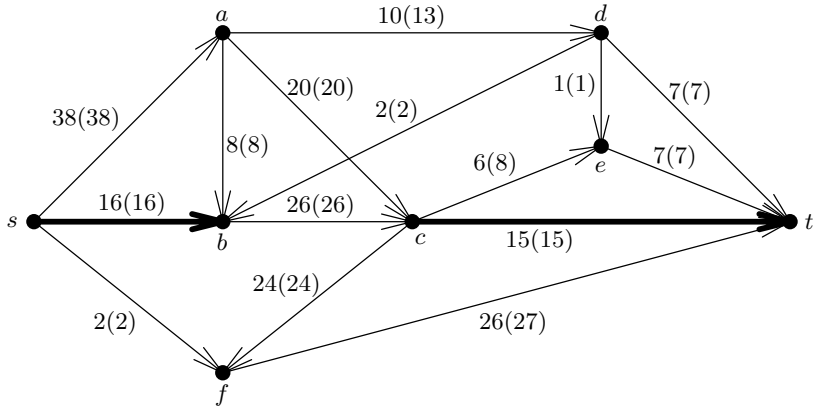


Fig. B.44. Flow h of value 56

by increasing the capacity of each of the edges sb, bc, and ct by τ. Note that there are other paths of cost 4 in the auxiliary network of Figure B.45, but the capacity of these paths is limited. We have now determined the cost function $z(v)$ completely (by executing the iteration step of the algorithm of Busacker and Gowen three times).

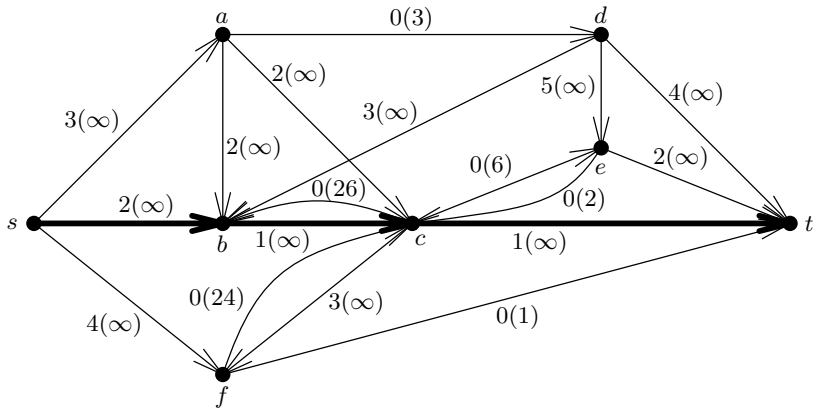


Fig. B.45. Auxiliary network for h

B.13 Solutions for Chapter 13

13.1.2 Let H be the graph which results from adding the edges of the complete graph on T to G. Clearly, G has a perfect matching if and only if H

does. Thus it suffices to show that condition (H) holds for G if and only if (T) holds for H. Put $n = |S| = |T|$.

First assume the validity of (H) for G. Given any subset X of V, we have to show $p(X) \leq |X|$. This is clear for $X = \emptyset$, as H is connected and contains precisely $2n$ vertices.

Next we consider the case where $X \subset T$ and $X \neq \emptyset$, and put $J = T \setminus X$. Then the components of $H \setminus X$ are the set $Y = J \cup \Gamma(J)$ and the singletons corresponding to the elements of $S \setminus \Gamma(J)$. If $|Y|$ is even, we have $p(X) = n - |\Gamma(J)|$ and $|X| = n - |J|$. Now (H) implies $|\Gamma(J)| \geq |J|$, so that $p(X) \leq |X|$ holds, as desired. If $|Y|$ is odd, $|\Gamma(J)| \geq |J|$ actually forces $|\Gamma(J)| \geq |J| + 1$, and the assertion follows in the same manner.

It remains to consider the case where X is not a subset of T. If $T \subset X$, the assertion holds trivially. Otherwise, let $X' = T \cap X$ and put $J = T \setminus X$, so that the components of $H \setminus X$ are the set $J \cup \Gamma(J)$ and the singletons corresponding to the elements of $S \setminus \Gamma(J)$. This implies

$$p(X) \;\leq\; p(X') + 1 \;\leq\; |X'| + 1 \;\leq\; |X|,$$

as required.

Conversely, assume the validity of condition (T) for H. Then one may check that (H) holds for G using a similar – actually easier – argument.

13.1.3 Let S be a subset of V, and denote the odd components of $G \setminus S$ by $V_1, \ldots, V_k$. Moreover, let m_i be the number of edges connecting a vertex in V_i to a vertex in S (for $i = 1, \ldots, k$). Since G does not contain any bridges, always $m_i \neq 1$. As G is 3-regular, $\sum_{v \in V_i} \deg v = 3|V_i|$ for $i = 1, \ldots, k$, so that

$$m_i \;=\; \sum_{v \in V_i} \deg v - 2|E_i|$$

is an odd number (where E_i denotes the edge set of the graph G_i induced on V_i). Hence always $m_i \geq 3$, which yields

$$p(S) \;=\; k \;\leq\; \frac{1}{3}(m_1 + \ldots + m_k) \;\leq\; \frac{1}{3}\Big(\sum_{v \in S} \deg v\Big) \;=\; |S|.$$

Thus condition (T) is satisfied, and the assertion follows from Theorem 13.1.1.

Figure B.46 shows a 3-regular graph containing bridges; this graph cannot have a perfect matching, as $p(\{v\}) = 3$. Finally, the Petersen graph defined in Exercise 1.5.10 is a 3-regular graph without bridges which does not admit a 1-factorization.

13.2.5 Let C be a Hamiltonian cycle in a graph G on $2n$ vertices. Choosing every other edge of C, we obtain a perfect matching of G. Thus every Hamiltonian graph having an even number of vertices admits a perfect

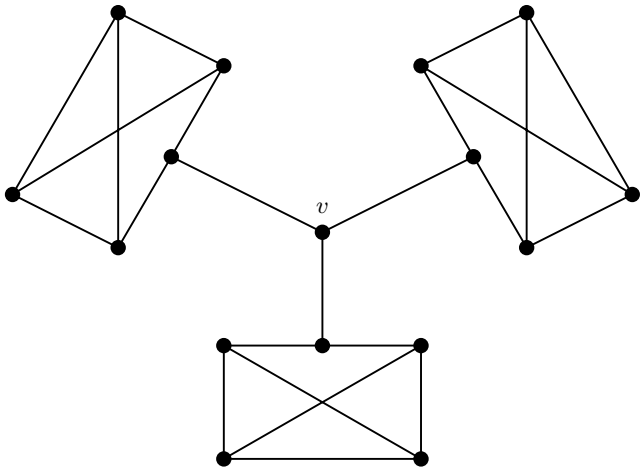


Fig. B.46. A 3-regular graph without a perfect matching

matching. Hence the two proposed criteria are indeed sufficient for the existence of a perfect matching of G, by Corollary 1.4.3 and Exercise 1.4.4.

13.2.6 We show first that G is 2-connected. Suppose otherwise. Then there exists a cut point v, so that $G \setminus v$ has two components X and Y. Choose an edge of the form vx with $x \in X$ and extend it to a perfect matching K. Then $|Y|$ has to be even and X has to be odd. However, the same argument also shows that $|Y|$ is odd and $|X|$ is even, a contradiction.

Assume first that G is bipartite, say $V = S \dot{\cup} T$. Suppose there are non-adjacent vertices $s \in S$ and $t \in T$, and let P be a path from s to t. As P has odd length, choosing the first, the third, ..., and the last edge of P gives a matching M. By hypothesis, M can be extended to a perfect matching M'. Then $M' \oplus P$ is a matching whose only exposed vertices are s and t. As s and t are not adjacent, this matching cannot be extended, a contradiction. Thus necessarily $G = K_{n,n}$ in the bipartite case.

It remains to consider the case where G is not bipartite. We show first that each vertex is contained in a cycle of odd length. Let v be a vertex of G, and let C be an arbitrary cycle of odd length; note that such a cycle exists by Theorem 3.3.5. We may assume that v is not contained in C. As G is 2-connected, Exercise 8.1.3 guarantees the existence of two paths P and P' with start vertex v and end vertex some vertex of C, which share only the vertex v. Thus these two paths together with the appropriate path in C which connects the two end vertices of P and P' form the desired cycle of odd length through v.

Now suppose that G contains two vertices u and v which are not adjacent. We claim that u and v are connected by a path of odd length. To see this, choose a cycle C of odd length containing v. If u is contained in C, the claim is clear. Otherwise, choose a path P with start vertex u and end vertex $w \neq v$ on C which does not contain any further vertices of C. Then P together with the appropriate path in C which connects w and v gives the required path of odd length from u to v. Now we obtain a contradiction just as in the bipartite case, and hence necessarily $G = K_{2n}$.

13.4.2 To simplify matters, we will make use of the inherent symmetry of the graph shown in Figure 13.9 and consider only its right half: we restrict attention to the subgraph G induced on the set $\{r, s, 1, 2, 3, 4, 5, 6\}$. The left half – which is isomorphic to the right half – can be treated in the same way, and a final augmenting path arises by joining the two individual augmenting paths via the matching edge ss'.

Beginning the procedure at r yields the alternating tree T shown in Figure B.47. When examining the edge 26, the blossom $B = \{2, 5, 6\}$ with base 2 is discovered and contracted. We obtain the graph $G' = G/B$ and the corresponding contracted tree $T' = T/B$ shown in Figure B.48.

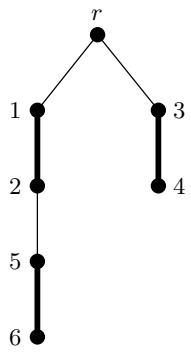


Fig. B.47. Alternating tree T for G

Next we examine the pseudovertex b and find the blossom $B' = \{r, 1, 3, 4, b\}$ with base r (because of edge 64). Contracting this blossom yields the graph $G'' = G'/B'$ which consists of the edge $b's$ only. This edge forms a trivial augmenting path P''. Expanding this path starting at s gives the augmenting path

$$P' : \quad s \text{ — } 1 \text{ — } b \text{ — } 4 \text{ — } 3 \text{ — } r$$

in G' and finally the augmenting path

$$P : \quad s \text{ — } 1 \text{ — } 2 \text{ — } 5 \text{ — } 6 \text{ — } 4 \text{ — } 3 \text{ — } r$$

in G.

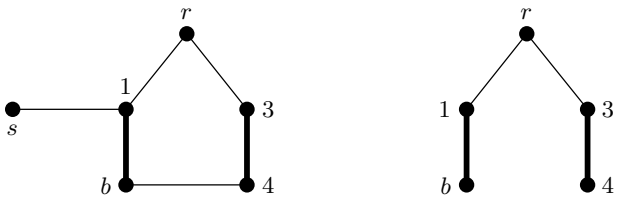


Fig. B.48. Contracted graph G/B with corresponding tree T/B

13.4.5 The graph G of Figure 13.19 is drawn again in Figure B.49. Obviously, $1 - 2 - 3 - 5$ is an augmenting path in G with respect to M. Contracting the blossom $B = \{2, 3, 4\}$, we obtain the graph $G' = G/B$ shown also in Figure B.49; this graph has the matching $M' = \{b6\}$. Clearly, G' does not contain an augmenting path with respect to M'.

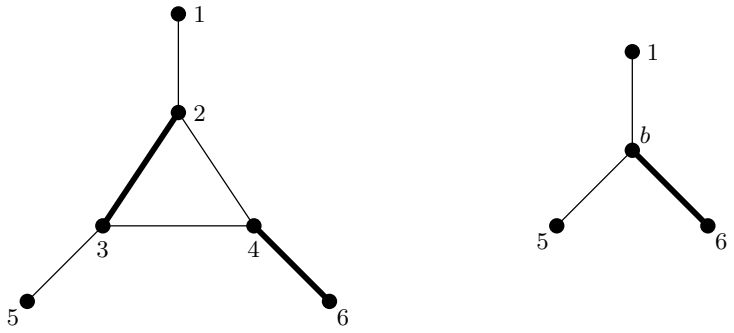


Fig. B.49. Graph G and contracted graph G/B

13.4.8 Let H be the graph with vertex set $V = \{1, \ldots, n\}$ which has an edge ij if and only if ij' (and then also ji') is an edge of G (for $i \neq j$). Then matchings in H consisting of k edges correspond to symmetric matchings in G with $2k$ edges. Thus a maximal symmetric matching of G can be determined using Algorithm 13.4.6 with complexity $O(n^3)$.[5]

13.5.3 Let G be the bipartite graph on $V = S \dot{\cup} T$ corresponding to $\mathbf{A} = (A_1, \ldots, A_n)$ (as defined in Section 7.3). Obviously, the partial transversals

[5] The work of Kocay and Stone and Fremuth-Paeger and Jungnickel mentioned at the beginning of this chapter uses the reverse approach: a symmetric bipartite graph G and an associated network are used for constructing a maximal matching in the corresponding graph H.

of $\mathbf{A}$ are precisely those subsets of S which are met by a matching of G. Therefore the partial transversals of $\mathbf{A}$ form a matroid by Corollary 13.5.2.

13.5.4 As the maximal matchings of G induce the bases of the matching matroid $(V, \mathbf{S})$, the assertion follows from Theorem 5.2.6. Alternatively, we may use Theorem 13.2.2: extending a matching using an augmenting path (as in the proof of Theorem 13.2.2) leaves any saturated vertex saturated, so that the assertion follows by induction.

B.14 Solutions for Chapter 14

14.1.2 Proceeding as outlined in Section 14.1, we obtain:

Procedure OPTMATCH$(n, w; M, D)$

(1) $W \leftarrow \max\{w_{ij} : i, j = 1, \ldots, n\}$;
(2) $V \leftarrow \{1, \ldots, n\} \cup \{1', \ldots, n'\} \cup \{s, t\}$;
(3) $E \leftarrow \{ij' : i, j = 1, \ldots, n\} \cup \{si : i = 1, \ldots, n\} \cup \{j't : j = 1, \ldots, n\}$;
(4) $G \leftarrow (V, E)$;
(5) **for** $i = 1$ **to** n **do**
(6) $\quad \gamma(si) \leftarrow 0$; $\gamma(i't) \leftarrow 0$; **for** $j = 1$ **to** n **do** $\gamma(ij') \leftarrow W - w_{ij}$ **od**
(7) **od**
(8) **for** $e \in E$ **do** $c(e) \leftarrow 1$ **od**
(9) OPTFLOW$(G, c, s, t, \gamma, n; f, \text{sol})$;
(10) $M \leftarrow \{ij' : f(ij') = 1\}$; $D \leftarrow \sum_{e \in M} w(e)$

To achieve a complexity of $O(n^3)$, we have to use the algorithm of Dijkstra for determining the shortest paths in step (7) of OPTFLOW, as explained in Section 10.5).

14.2.6 During the first four phases, we obtain (without any changes) the edges $\{1, 4'\}$, $\{2, 9'\}$, $\{3, 6'\}$, and $\{4, 1'\}$ (in this order). Even the feasible node weighting $(\mathbf{u}, \mathbf{v})$ remains unchanged.

During the fifth phase (where $i = 5$), the only vertex j' with $\delta_j = 0$ is $9'$, which is saturated already. Nothing is changed by $i = 2$, because mate$(9') = 2$ is the smallest vertex in Q. Next, for $i = 6$, we find the edge $\{6, 3'\}$. Similarly, during the phases 6 and 7, the edges $\{7, 8'\}$ and $\{8, 7'\}$ are constructed. Up to this point, $(\mathbf{u}, \mathbf{v})$ was not changed.

During phase 8, we have $i = 5$, $i = 2$ (because of $\delta_9 = 0$, mate$(9') = 2$), $i = 9$, and $i = 7$ (because of $\delta_8 = 0$, mate$(8') = 7$). Now $J = \{2, 5, 7, 9\}$, $K = \{8', 9'\}$, and $\delta = 1$, so that the u_i and v_j have to be changed. We obtain the exposed vertex $5'$ with $\delta_5 = 0$, and the edge $\{9, 5'\}$ is added to the matching constructed so far.

The ninth (and last) phase is the most involved one. Again, we first have $i = 5$, $i = 2$, and $i = 7$. Then $(\mathbf{u}, \mathbf{v})$ has to be changed according to $J = \{2, 5, 7\}$, $K = \{8', 9'\}$, and $\delta = 2$. Then $\delta_4 = 0$ and mate $(4') = 1$, so that $i = 1$.

Again, $(\mathbf{u}, \mathbf{v})$ has to be changed, this time for $J = \{1, 2, 5, 7\}$, $K = \{4', 8', 9'\}$ and $\delta = 3$. Three more changes of $(\mathbf{u}, \mathbf{v})$ follow: for $J = \{1, 2, 4, 5, 7\}$, $K = \{1', 4', 8', 9'\}$, $\delta = 1$; $J = \{1, 2, 4, 5, 7, 9\}$, $K = \{1', 4', 5', 8', 9'\}$, $\delta = 2$; and $J = \{1, 2, 4, 5, 7, 8, 9\}$, $K = \{1', 4', 5', 7', 8', 9'\}$, $\delta = 5$. Now $2'$ is exposed and we can complete the matching by adding the edge $\{5, 2'\}$.

We show the values for $(\mathbf{u}, \mathbf{v})$ below; the entries corresponding to edges used in the construction are in bold type. Note that indeed $w(M) = \sum(u_i + v_i)$ $(= 603)$ holds.

$$\begin{pmatrix} 0 & 31 & 24 & \mathbf{80} & 62 & 39 & 24 & 41 & 42 \\ 31 & 0 & 0 & 34 & 54 & 5 & 51 & 45 & \mathbf{61} \\ 24 & 0 & 0 & 31 & 32 & \mathbf{59} & 28 & 44 & 25 \\ \mathbf{80} & 34 & 31 & 0 & 65 & 45 & 25 & 44 & 47 \\ 62 & \mathbf{54} & 32 & 65 & 0 & 38 & 48 & 66 & 68 \\ 39 & 5 & \mathbf{59} & 45 & 38 & 0 & 8 & 25 & 18 \\ 24 & 51 & 28 & 25 & 48 & 8 & 0 & \mathbf{71} & 66 \\ 41 & 45 & 44 & 44 & 66 & 25 & \mathbf{71} & 0 & 69 \\ 42 & 61 & 25 & 47 & \mathbf{68} & 18 & 66 & 69 & 0 \end{pmatrix} \begin{matrix} 69 \\ 47 \\ 59 \\ 72 \\ 54 \\ 59 \\ 57 \\ 66 \\ 61 \end{matrix}$$

$$\begin{matrix} 8 & 0 & 0 & 11 & 7 & 0 & 5 & 14 & 14 \end{matrix} \quad \mathbf{v} \backslash \mathbf{u}$$

Note that the matching consisting of the edges $\{1, 4'\}$, $\{2, 5'\}$, $\{3, 6'\}$, $\{4, 1'\}$, $\{5, 9'\}$, $\{6, 3'\}$, $\{7, 8'\}$, $\{8, 7'\}$, and $\{9, 2'\}$ is optimal as well.

14.2.7 The algebraic assignment problem for the ordered semigroup $(\mathbb{R}_0^+, \min)$ yields the bottleneck assignment problem.

14.2.8 During the first two phases, the edges $\{1, 3'\}$ and $\{2, 4'\}$ are found. In phase 3, first $i = 3$ and $\delta_4 = 1$; as $\text{mate}(4') = 2$, then $i = 2$, and we find the exposed vertex $5'$ with $\delta_5 = 1$. Thus the present matching is changed using $p(5) = 2$, $\text{mate}(2) = 4'$, and $p(4) = 3$; we obtain the edges $\{1, 3'\}$, $\{2, 5'\}$, and $\{3, 4'\}$.

In phase 4, the current matching is enlarged by the edge $\{5, 2'\}$. During the final phase, $(\mathbf{u}, \mathbf{v})$ has to be changed twice: first with $J = \{2, 3, 4\}$, $K = \{4', 5'\}$, and $\delta = 5/4$; and then with $J = \{2, 3, 4, 5\}$, $K = \{2', 4', 5'\}$, and $\delta = 36/35$. The matching is changed once again; we get the solution

$$M = \{\{1, 3'\}, \{2, 5'\}, \{3, 1'\}, \{4, 4'\}, \{5, 2'\}\}.$$

The corresponding entries are set bold in the matrix below. We also check our calculations: $w(M) = \prod(u_i, v_i) = 15120$.

$$\begin{pmatrix} 3 & 8 & \mathbf{9} & 1 & 6 \\ 1 & 4 & 1 & 5 & \mathbf{5} \\ \mathbf{7} & 2 & 7 & 9 & 2 \\ 3 & 1 & 6 & \mathbf{8} & 8 \\ 2 & \mathbf{6} & 3 & 6 & 2 \end{pmatrix} \begin{matrix} 9 \\ 35/9 \\ 7 \\ 56/9 \\ 35/6 \end{matrix}$$

$$1 \quad \tfrac{36}{35} \quad 1 \quad \tfrac{9}{7} \quad \tfrac{9}{7} \qquad \mathbf{v}\backslash\mathbf{u}$$

Note that this product-optimal matching accidentally coincides with the optimal matching of Example 14.2.5; as Exercise 14.2.10 shows, this really is exceptional.

14.2.9 Denote the given weight matrix by $W = (w_{ij})$ and put $W' = (\log w_{ij})$. Then the product-optimal matchings with respect to W are precisely the optimal matchings with respect to W'.

However, this transformation is not of practical interest. When executing calculations with W' using a computer, errors occur because of rounding (logarithms are irrational in general), and this means we cannot check our solution by comparing $w'(M)$ with $\sum(u'_i + v'_i)$.

Alternatively, we might consider doing all calculations *symbolically*, so that we perform operations such as replacing $\log p + \log q$ with $\log pq$. But then we may as well use the version of the Hungarian algorithm modified for $(\mathbb{R}^+, \cdot)$. Nevertheless, the above transformation at least yields an immediate proof for the correctness of this approach.

14.2.10 For the matrix

$$\begin{pmatrix} 3 & \mathbf{1} & 1 \\ 1 & 4 & \mathbf{5} \\ \mathbf{6} & 1 & 4 \end{pmatrix},$$

the matching given by the bold entries has weight $12 = 1 + 5 + 6$ and is obviously optimal. However, it is not product-optimal. On the other hand, the matching corresponding to the entries in the main diagonal is product-optimal but not optimal.

14.3.2 First let A be the incidence matrix of a digraph G. By Lemma 10.3.1, the vector $\mathbf{f} = (f_e)_{e \in E}$ gives a circulation if and only if $A\mathbf{f}^T = 0$. Therefore we get the ILP

$$\text{minimize } \gamma\mathbf{x}^\mathbf{T} \text{ subject to } A\mathbf{x}^T = \mathbf{0}^T,\ \mathbf{b} \le \mathbf{x} \le \mathbf{c}.$$

For the second part, let T be a spanning tree of a graph G on n vertices. As T contains $n-1$ edges and is acyclic, we can use the following ZOLP:

$$\text{maximize } \mathbf{w}\mathbf{x}^\mathbf{T} \text{ subject to } \mathbf{1}\mathbf{x}^T = n-1 \text{ and } \sum_{e \in C} x_e \le |C| - 1,$$

where $\mathbf{x} = (x_e)_{e \in E}$ and where C runs over all cycles in G. This approach is not interesting in practice, as the ZOLP will (in general) contain far too many inequalities.

14.4.6 Let G be a regular bipartite multigraph with vertex set $V = S \dot{\cup} T$, where $|S| = |T| = n$. We define the *adjacency matrix* $A = (a_{ij})_{i,j=1,\ldots,n}$ of the multigraph G as follows: a_{ij} is the number of edges of G with end vertices i and j', where we assume $S = \{1, \ldots, n\}$ and $T = \{1', \ldots, n'\}$.

Thus A is a matrix with nonnegative integral entries, and its row and column sums are constant. By Theorem 7.4.5, we can write A as a sum of permutation matrices. As each permutation matrix corresponds to a 1-factor of G, the decomposition of A yields a 1-factorization of G.

14.4.7 Obviously, $\mathbf{L}(G) \subset \mathbf{H}(G) \cap \mathbb{Z}^E$. Now let $\mathbf{x}$ be a vector in $\mathbf{H}(G) \cap \mathbb{Z}^E$, and choose some positive integer k which is larger than the absolute value of $\mathbf{x}$. Then $\mathbf{x}' = \mathbf{x} + \sum k\mathbf{m}$ is likewise an element of $\mathbf{H}(G) \cap \mathbb{Z}^E$, where $\mathbf{m}$ runs over the incidence vectors of the perfect matchings of G. Moreover, $\mathbf{x}' \geq \mathbf{0}$.

We now define a regular bipartite multigraph G' by replacing each edge e of G with x'_e parallel edges. Note that G' is indeed regular, since $\mathbf{x}'$ is contained in $\mathbf{H}(G)$. By Exercise 14.4.6, G' can be decomposed into 1-factors. As each 1-factor of G' induces a 1-factor of G, we see that $\mathbf{x}'$ has to be a linear combination of incidence vectors of perfect matchings of G with nonnegative integral coefficients, so that also $\mathbf{x} = \mathbf{x}' - \sum k\mathbf{m}$ is contained in $\mathbf{L}(G)$.

14.5.6 Every closed walk of G which contains each edge at least once induces a circulation $\mathbf{f}$ on G: define f_e as the number of times e occurs in the given walk. Note $\mathbf{f} \geq \mathbf{1}$.

Conversely, every circulation $\mathbf{f}$ with $\mathbf{f} \geq \mathbf{1}$ induces a closed walk on G which contains all edges: replace each edge e with f_e parallel edges. By Theorem 1.6.1, the resulting pseudosymmetric digraph contains an Euler tour, which induces the desired walk.

Note that G is obviously connected. Thus a shortest directed closed walk corresponds to an optimal circulation with respect to the capacity constraints $b(e) = 1$ and $c(e) = \infty$ and the cost $\gamma(e) = w(e)$ (for all $e \in E$). Thus the directed CPP can be solved using the algorithm OPTCIRC from Section 10.7.

14.6.5 $W\colon s - c - t$ is a shortest $\{s,t\}$-path (of length -1). The corresponding f-factors are

$$F = \{\{a,a\},\{b,b\},sc,ct\} \quad \text{and} \quad F' = \{\{a,a\},\{b,b\},sc,ct,c_gb_g,a_eb_e\}\,,$$

and the corresponding perfect matching is $M = \{a'a'', b'b'', sc', c''t, a_eb_e, c_gb_g\}$.

14.6.6 If (G,w) contains cycles of negative length, the method is not applicable, as the proof of Lemma 14.6.1 shows. (The construction would yield

a path from s to t together with – possibly negative – cycles.) This is not surprising, since the problem of finding a shortest path is then NP-hard; see Appendix A or [GaJo79].

14.6.9 Use the following modification of the transformation described in Theorem 14.6.7: each edge of the form tv is now replaced by an edge tv'' (instead of tv'), and an eventual edge st is removed.
For the graph G of Example 14.6.8, the even $\{s,t\}$-path

$$W: \quad s \text{ — } u \text{ — } v \text{ — } a \text{ — } t$$

corresponds to the perfect matching

$$M = \{su', u''v'', v'a', a''t, b'b'', c'c''\}$$

in the auxiliary graph G''; see Figures B.50 and B.51.

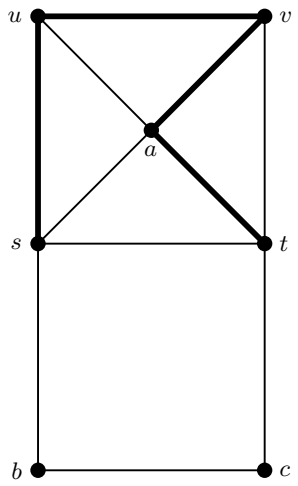


Fig. B.50. A path of even length in G

14.7.2 Define the weight function w on H as follows:

$$w_e = \begin{cases} 0 & \text{for } e \notin E \\ c-1 & \text{for } e \in E_1 \\ c & \text{for } e \in E \setminus E_1. \end{cases}$$

Now let M be an optimal matching of H with respect to this weight function. Then M consists exclusively of edges of G if and only if $w(M) \geq n(c-1)$, where $|V| = 2n$. In this case, the number of edges of M contained in E_1 is $cn - w(M)$. Thus the problem RPM1 has a solution if and only if $cn - w(M) \leq b_1$.

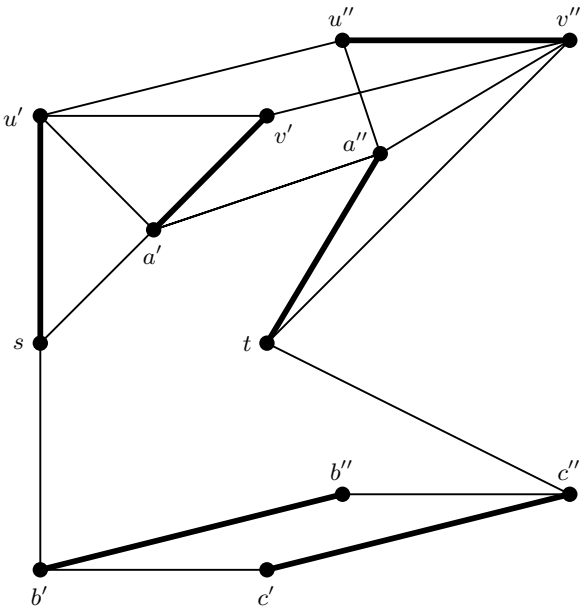


Fig. B.51. The corresponding perfect matching in G''

By Result 14.4.5, $w(M)$ can be determined with complexity $O(n^3)$, and the assertion follows.

14.7.4 Put $E_1 = R$, $E_2 = E \setminus R$, $b_1 = b$, and $b_2 = n - b$ (where $|V| = 2n$). The perfect matchings of G which satisfy condition (14.10) of Problem 14.7.1 for these values are precisely the desired solutions of EPM, since every perfect matching contains exactly n edges.

14.8.6 We use the edge labelling of the Petersen graph given in Figure 14.15 for indexing the coordinate positions, so that the edge labelled i corresponds to the coordinate position i (for $i = 1, \ldots, 15$). As noted in the hint, it clearly suffices to check that the cyclic shifts of the codewords in some basis of P^* are again in P^*.

Thus we begin by determining a basis of the Petersen code P, using the method given in the proof of Theorem 10.3.6. We first have to select a spanning tree T for the Petersen graph; let us choose the five spoke edges 3, 6, 9, 12, and 15 together with four edges of the outer cycle, say 2, 5, 8, and 11. Then each of the six edges $e \notin T$ determines a unique cycle contained in $T \cup \{e\}$, namely:

- $C_1 = \{1, 15, 2, 11, 5, 12\}$, belonging to $1 \notin T$;
- $C_4 = \{4, 15, 2, 11, 3\}$, belonging to $4 \notin T$;

- $C_7 = \{7, 6, 8, 2, 11, 3\}$, belonging to $7 \notin T$;
- $C_{10} = \{10, 9, 2, 8, 6\}$, belonging to $10 \notin T$;
- $C_{13} = \{13, 9, 11, 5, 12\}$, belonging to $13 \notin T$;
- $C_{14} = \{14, 8, 2, 11, 5\}$, belonging to $14 \notin T$.

Then we obtain the augmented Petersen code P^* by adjoining an odd subgraph, for instance, the matching formed by the five spoke edges as a further basis element, see Example 10.11.11:

- $S = \{3, 6, 9, 12, 15\}$.

Now it suffices to check that the cyclic shifts of these seven codewords are again in P^*. We obtain the following images, where τ denotes the cyclic shift:

- $C_1^\tau = \{1, 2, 3, 6, 12, 13\}$, an odd subgraph;
- $C_4^\tau = \{1, 3, 4, 5, 12\}$, a cycle;
- $C_7^\tau = \{3, 4, 7, 8, 9, 12\}$, an odd subgraph;
- $C_{10}^\tau = \{3, 7, 9, 10, 11\}$, a cycle;
- $C_{13}^\tau = \{6, 10, 12, 13, 14\}$, a cycle;
- $C_{14}^\tau = \{3, 6, 9, 12, 15\}$, the spoke matching;
- $S^\tau = \{1, 4, 7, 10, 13\}$, the inner pentagram.

All these words indeed belong to P^*.

B.15 Solutions for Chapter 15

15.2.2 Consider an edge e of smallest weight in a perfect matching M of minimal weight, say $e = ij$ with weight w_{ij}. In the symmetric case, $w_{ij} = w_{ji}$. If the remaining weights are much larger, also the edge ji will belong to M. Hence it is more likely that a pair of antiparallel edges occurs in the symmetric case than in the asymmetric case.

15.2.5 Let T be the minimal spanning tree associated with the given TSP which is shown in Figure 15.2. Note that we obtain a minimal s-tree for $s = Aa$ by adding the edge $AaFr$, so that the weight is $186 + 26 = 212$. Similarly, we obtain weights of $186 + 34 = 220$ and $186 + 22 = 208$ for for $s = Ba$ and $s = Mu$, respectively.

For $s = Du$, a minimal spanning tree on the remaining eight vertices consists of the edges $BeHa$, $HaAa$, $AaFr$, $FrSt$, $StBa$, $StNu$ and $NuMu$, so that the weight of a minimal s-tree is $187 + 8 + 23 = 218$.

For the remaining four choices of s, we just list the weight of a minimal s-tree: $184 + 20 + 22 = 226$ for $s = Fr$; $158 + 29 + 43 = 230$ for $s = Ha$; $172 + 17 + 19 = 208$ for $s = Nu$; and $176 + 19 + 20 = 215$ for $s = St$.

15.2.6 Let T be a minimal spanning tree associated with the given TSP, and assume that s is a leaf of T. Then we obtain a minimal s-tree by adding an

edge of smallest weight among all edges not in T and incident with s to T. This follows by observing that $T \setminus e$, where e is the unique edge of T incident with s, has to be a minimal spanning tree for the complete graph induced on the remaining points.

In general, however, matters cannot be as simple. For instance, if s has degree at least 3 in T, we cannot obtain an s-tree containing T for trivial reasons. This observation also suggests examples showing that the strategy for selecting s which we have used in Example 15.2.4 may fail badly.

For instance, let T be a star for which all edges have weight a, and assume that all remaining edges have weight $b > a$; note that T is the unique minimal spanning tree for TSP-instances of this type. Our strategy does not allow us to select s as the center of T, which would lead to a minimal s-tree of weight $(n-2)b+2a$. Any other choice of s is permissible and would result in a minimal s-tree of weight $(n-2)a+2b$; in general, this will be a considerably smaller – and hence inferior – bound. Thus our strategy can prevent the optimal choice for s, and the deviation between the resulting bounds can even be made arbitrarily large.

15.2.7 By Corollary 1.2.11, there are $(n-1)^{n-3}$ distinct spanning trees on the remaining $n-1$ vertices. To each of these trees, we have to add one of the $(n-1)(n-2)/2$ pairs of edges incident with s, so that the total number of s-trees of K_n is $\frac{1}{2}(n-2)(n-1)^{n-2}$.

15.2.8 In every tour, each vertex i is incident with two edges whose weight is at least $s(i)+s'(i)$. This leads to the desired inequality and yields a lower bound of 214 for the TSP of Example 15.1.2; note that w is integral.

15.3.1 Let B be an s-tree. Then

$$\sum_i p_i \;=\; c \times \sum_i (\deg_B i - 2) \;=\; c \times \Big(\sum_i \deg_B i \;-\; 2n \Big) \;=\; c(2n-2n) \;=\; 0.$$

15.5.4 As in Example 15.5.3, we begin with $s = Fr$. In the first two steps, we obtain the partial tour (Fr, St, Fr) with length 40 and then (Fr, St, Nu, Fr) with length 61.

Now Mu is inserted, and we get (Fr, St, Mu, Nu, Fr) with length 81. The next iteration yields (Fr, Du, St, Mu, Nu, Fr) with length 125. Inserting Aa between Du and St yields a partial tour with length 138.

We proceed with $(Fr, Du, Aa, Ba, St, Mu, Nu, Fr)$ with length 176; after this, we insert Ha between Fr and Du, which yields a partial tour with length 246. Finally, we obtain the tour $(Fr, Be, Ha, Du, Aa, Ba, St, Mu, Nu, Fr)$ with length 281.

15.6.4 First, the edges $AaMu$ and $FrBe$ are replaced with $MuBe$ and $AaFr$. This reduces the weight of the tour shown in Figure 15.7 by

$34 = (64 + 56) - (60 + 26)$; the resulting tour of weight $307 - 34 = 273$ is $(Aa, Du, Ha, Be, Mu, Nu, Ba, St, Fr, Aa)$.

Next, the edges $NuBa$ and $FrSt$ are replaced with $BaFr$ and $StNu$. This yields the tour $(Aa, Du, Ha, Be, Mu, Nu, St, Ba, Fr, Aa)$ and reduces the weight by $(43 + 20) - (34 + 19) = 10$ to 263.

Finally, we replace the edges $StNu$ and $BeMu$ with $StMu$ and $NuBe$, which yields the (optimal) tour of length 250 shown in Figure 15.9; indeed, this step reduces the weight by $(19 + 60) - (22 + 44) = 13$.

15.7.7 As $\mathbf{A}'$ is assumed to be an ε-approximative algorithm for TSP, we can use it to solve the problem HC as described in the proof of Theorem 15.4.1. Note that each iteration of the algorithm $\mathbf{A}'$ (that is, each application of $\mathbf{A}$ to a neighborhood $N(f)$) decreases the weight of the current tour – with the exception of the final application of $\mathbf{A}$, which merely discovers that the current tour is now locally optimal.

As the weight function defined in the proof of Theorem 15.4.1 takes only two values, there can be only $n + 1$ distinct lengths of tours. Therefore $\mathbf{A}'$ cannot need more than $O(n)$ iterations of $\mathbf{A}$. Since $\mathbf{A}$ is polynomial, $\mathbf{A}'$ would be a polynomial algorithm for HC, so that P=NP by Result 13.2.2.

15.7.8 Suppose that the given problem could be solved polynomially. We show that this implies that we can find even an optimal tour in polynomial time. We may assume that all weights are ≥ 2. (If necessary, we add a constant to all weights.)

Now we check whether some specified edge e_1 is contained in an optimal tour. If the answer is yes, we reduce $w(e_1)$ to 1; this ensures that e_1 has to be contained in *every* optimal tour for the modified problem. More precisely, the optimal tours for the new problem are precisely those optimal tours for the old problem which contain the edge e_1. Continuing in this manner, we obtain an optimal tour for the original problem after $O(n^2)$ calls of the decision problem which we assumed to be polynomial.

15.7.9 Note that TSP suboptimality does not actually *produce* a better tour: it only tells us if such a tour *exists*. Hence it is not possible to apply a hypothetical polynomial algorithm $\mathbf{A}$ for this problem repeatedly, which would be necessary if we were to use $\mathbf{A}$ to construct a polynomial algorithm for HC.

C

List of Symbols

Now pray, what did he mean by that?
Richard Brinsley Sheridan

C.1 General Symbols

This first part of the list contains some general symbols which are more or less standard. The special symbols of graph theory will be covered in the next section.

Sets

$A \cup B$	union of the sets A and B
$A \dot\cup B$	disjoint union of the sets A and B
$A \cap B$	intersection of the sets A and B
$A \times B$	Cartesian product of the sets A and B
$A \setminus B$	A without B: $A \cap \overline{B}$
$A \oplus B$	symmetric difference of A and B: $(A \setminus B) \cup (B \setminus A)$
2^A	power set of A
$\overline{A}$	complement of A (with respect to a given universal set)
A^t	set of ordered t-tuples with elements from A
$\binom{A}{t}$	set of t-subsets of A
$\|A\|$	cardinality of A
$\emptyset$	empty set
$A \subset B$	A is a subset of B

Mappings

$f\colon A \to B$	f is a mapping from A to B
$f(x)$	image of x under the mapping f
$f\colon x \mapsto y$	f maps x to y: $f(x) = y$
$f(X)$	$\{f(x)\colon x \in X\}$ for $f\colon A \to B$ and $X \subset A$
$\operatorname{supp} f$	support of f

Numbers

$\sum_{i=1}^{n} a_i$	$a_1 + \ldots + a_n$
$\prod_{i=1}^{n} a_i$	$a_1 a_2 \ldots a_n$
$\lceil x \rceil$	smallest integer $\geq x$ (for $x \in \mathbb{R}$)
$\lfloor x \rfloor$	largest integer $\leq x$ (for $x \in \mathbb{R}$)
$n!$	$n(n-1)(n-2)\ldots 1$ (for $n \in \mathbb{N}$)
$\binom{n}{t}$	number of t-subsets of an n-set
e	base of the natural logarithm

Matrices

A^T	transpose of the matrix A
J	matrix with all entries 1
I	identity matrix
$\operatorname{diag}(a_1, \ldots, a_n)$	diagonal matrix with entries $a_{11} = a_1, \ldots, a_{nn} = a_n$
(a_{ij})	matrix with entries a_{ij}
$\det A$, $\lvert A \rvert$	determinant of the matrix A
$\operatorname{per} A$	permanent of the matrix A

Sets of numbers and algebraic structures

$\mathbb{N}$ or $\mathbb{Z}^+$	set of natural numbers (not including 0)
$\mathbb{N}_0$	set of natural numbers including 0
$\mathbb{Z}$	ring of integers

$\mathbb{Z}_n$	ring of integers modulo n
$\mathbb{Q}$	field of rational numbers
$\mathbb{Q}^+$	set of positive rational numbers
$\mathbb{Q}_0^+$	set of non-negative rational numbers
$\mathbb{R}$	field of real numbers
$\mathbb{R}^+$	set of positive real numbers
$\mathbb{R}_0^+$	set of non-negative real numbers
K^*	multiplicative group of the field K
K^n	n-dimensional vector space over the field K
$K^{(n,n)}$	ring of $(n \times n)$-matrices over the field K
S_n	symmetric group acting on n elements

Miscellaneous

$x := y$, $y =: x$	x is defined to be y
$x \leftarrow y$	x is assigned the value of y

C.2 Special Symbols

This second part of the list contains symbols from graph theory and the symbols introduced in this book.

Graphs and networks

$\overline{G}$	complementary graph of the graph G
G_{red}	reduced digraph of the acyclic digraph G
$G\|U$	subgraph of G induced on the vertex set U
$G \setminus e$	G with the edge e discarded
$G \setminus T$	subgraph of G induced on the set $V \setminus T$
$G \setminus v$	subgraph of G induced on the set $V \setminus \{v\}$ (for $v \in V$)
G/B	contraction of the graph G with respect to the blossom B

G/e	contraction of the graph G with respect to the edge e
$\|G\|$	multigraph underlying the directed multigraph G
(G)	underlying graph for the multigraph G
$\overrightarrow{G}$	complete orientation of the graph G
$[G]$	closure of the graph G
$bc(G)$	block-cutpoint graph of G
$G(H, S)$	Cayley graph defined b the group H and the set S
$G_{s,n}$	de Bruijn graph
$H_{\mathbf{u},\mathbf{v}}$	equality subgraph for G with respect to $(\mathbf{u}, \mathbf{v})$
K_n	complete graph on n vertices
$K_{m,n}$	complete bipartite graph on $m + n$ vertices
$L(G)$	line graph of G
N', $N'(f)$	auxiliary network for N (with respect to the flow f)
N'', $N''(f)$	layered auxiliary network for N (with respect to the flow f)
$T(G)$	tree graph of the connected graph G
T_n	triangular graph on $\binom{n}{2}$ vertices

Objects in graphs

$C_T(e)$	cycle determined by the spanning tree T and the edge $e \notin T$
$a \overset{e}{\text{—}} b$	edge $e = ab$
e^-	the start vertex (*tail*) of the edge e
e^+	the end vertex (*head*) of the edge e
$E(S)$, $E(X, Y)$	edge set corresponding to the cut $S = (X, Y)$
$E\|V'$	edge set induced on the vertex set V'
F_ε	set of ε-fixed edges (with respect to a given circulation)
$S_T(e)$	cut determined by the spanning tree T and the edge $e \in T$
$\Gamma(J)$	neighborhood of the vertex set J
$\Gamma(v)$	neighborhood of the vertex v

Parameters for graphs

$\mathrm{ch}(G)$	choosability of G

$\deg v$	degree of the vertex v
$d_{\text{in}}(v)$	indegree of the vertex v
$d_{\text{out}}(v)$	outdegree of the vertex v
g	girth of G
n_d	number of vertices of degree d
$\alpha(G)$	independence number of G
$\alpha'(G)$	maximal cardinality of a matching of G
$\beta(G)$	minimal cardinality of a vertex cover of G
$\delta(G)$	minimal degree of a vertex of G
$\Delta(G)$	maximal degree of a vertex of G
Δ	minimal number of paths in a dissection of G (*Dilworth number*)
$\theta(G)$	clique partition number of G
$\kappa(G)$	connectivity of G
$\lambda(G)$	edge connectivity of G
$\nu(G)$	cyclomatic number of G
$\chi(G)$	chromatic number of G
$\chi'(G)$	chromatic index of G
$\omega(G)$	maximal cardinality of a clique in G

Mappings on graphs and networks

$a(x)$	supply at the vertex x
$b(e)$	lower capacity constraint for the edge e
$c(e)$	capacity of the edge e
$c(W)$	capacity of the path W
$c(S,T)$	capacity of the cut (S,T)
$d(x)$	demand at the vertex x
$d(a,b)$	distance between the vertices a and b
$d_H(a,b)$	distance between vertices a and b in the graph H
$f(S,T)$	flow value for the cut (S,T) with respect to the flow f
$m(K)$	mean weight of the cycle K

$p(v)$	flow potential at the vertex v
$p(S)$	number of odd components of $G \setminus S$
$r(v)$	rank of the vertex v in an acyclic digraph
$w(e)$	weight (or length) of the edge e
$w(f)$	value of the flow f
$w(P)$	weight of the optimal solution for the problem P
$w(X)$	weight (or length) of a set X of edges
$w(\pi)$	weight of the tour π
$w_A(P)$	weight of the solution for problem P determined by algorithm A
$w(s,t)$	value of a maximal flow between s and t (in a symmetric network)
$\gamma(e)$	cost of the edge e
$\gamma^{(\varepsilon)}(f)$	cost of the edge e increased by ε
$\gamma_p(e)$	reduced cost of the edge e (with respect to the potential p)
$\gamma(f)$	cost of the circulation or the flow f
$\gamma(M)$	cost of the perfect matching M
$\gamma(v)$	cost of an optimal flow with value v
δq	potential difference
$\varepsilon(f)$	optimality parameter for the circulation f
$\epsilon(v)$	excentricity of the vertex v
$\kappa(s,t)$	maximal number of vertex disjoint paths from s to t
$\lambda(s,t)$	maximal number of edge disjoint paths from s to t
$\mu(G,w)$	minimum cycle mean in the network (G,w)
$\pi_V(T)$	Prüfer code of the tree T on the vertex set V

Matroids and independence systems

$\mathrm{lr}(A)$	lower rank of the set A
$M(G)$	graphic matroid corresponding to G
M^*	dual matroid of the matroid M
$\overline{M}$	hereditary closure of the set system M

rq (M)	rank quotient of the independence system M
ur (A)	upper rank of the set A
$\mathbf{S}\|N$	restriction of the set system $\mathbf{S}$ to the set N
$\rho(A)$	rank of the set A
$\sigma(A)$	span of the set A

Matrices

A	adjacency matrix of a graph
A'	degree matrix of a graph
M	incidence matrix of a graph or a digraph
$\rho(A)$	term rank of the matrix A

Codes

$C_E(G)$	even graphical code based on G
$d(\mathbf{x}, \mathbf{y})$	Hamming distance of $\mathbf{x}$ and $\mathbf{y}$
d	minimum distance of a code
k	dimension of a linear of a code
n	length of a code
$w(\mathbf{x})$	weight of $\mathbf{x}$

Miscellaneous

A_v	adjacency list for the vertex v
A'_v	reverse adjacency list for the vertex v
$d(\mathbf{a})$	deficiency of the set family $\mathbf{A}$
$t(\mathbf{A})$	transversal index of the set family $\mathbf{A}$
$O(f(n))$	upper bound on the complexity
$\Omega(f(n))$	lower bound on the complexity
$\Theta(f(n))$	rate of growth

References

Round up the usual suspects.
From 'Casablanca'

[AaLe97] Aarts, E., Lenstra, J.K.: *Local Search in Combinatorial Optimization.* Wiley, New York (1997)

[Abu90] Abu-Sbeih, M.Z.: On the number of spanning trees of K_n and $K_{m,n}$. *Discr. Math.* **84**, 205–207 (1990)

[AhHU74] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: *The Design and Analysis of Computer Algorithms.* Addison Wesley, Reading, Mass. (1974)

[AhHU83] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: *Data Structures and Algorithms.* Addison Wesley, Reading, Mass. (1983)

[AhGOT92] Ahuja, R.K., Goldberg, A.V., Orlin, J.B., Tarjan, R.E.: Finding minimum-cost flows by double scaling. *Math. Progr.* **53**, 243–266 (1992)

[AhKMO92] Ahuja, R.K., Kodialam, M., Mishra, A.K., Orlin, J.B.: Computational testing of maximum flow algorithms. Sloan working paper, Sloan School of Management, MIT (1992)

[AhMO89] Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network flows. In: Nemhauser, G.L., Rinnooy Kan, A.H.G., Todd, M.J. (eds) *Handbooks in Operations Research and Management Science, Vol 1: Optimization*, pp. 211–369. North Holland, Amsterdam (1989)

[AhMO91] Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Some recent advances in network flows. *SIAM Review* **33**, 175–219 (1991)

[AhMO93] Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, Englewood Cliffs, N.J. (1993)

[AhMOT90] Ahuja, R.K., Mehlhorn, K., Orlin, J.B., Tarjan, R.E.: Faster algorithms for the shortest path problem. *J. Ass. Comp. Mach.* **37**, 213–223 (1990)

[AhOr89] Ahuja, R.K., Orlin, J.B.: A fast and simple algorithm for the maximum flow problem. *Oper. Res.* **37**, 748–759 (1989)

[AhOr92] Ahuja, R.K., Orlin, J.B.: The scaling network simplex algorithm. *Oper. Res.* **40**, Suppl. 1, S5–S13 (1992)

[AhOr95] Ahuja, R.K., Orlin, J.B.: A capacity scaling algorithm for the constrained maximum flow problem. *Networks* **25**, 89–98 (1995)

[AhOST94] Ahuja, R.K., Orlin, J.B., Stein, C., Tarjan, R.E.: Improved algorithm for bipartite network flow. *SIAM J. Computing* **23**, 906–933 (1994)

[AhOT89] Ahuja, R.K., Orlin, J.B., Tarjan, R.E.: Improved time bounds for the maximum flow problem. *SIAM J. Comp.* **18**, 939–954 (1989)

[Aig84] Aigner, M.: *Graphentheorie. Eine Entwicklung aus dem 4-Farben-Problem.* Teubner, Stuttgart (1984)

[Aig97] Aigner, M.: *Combinatorial Theory.* Springer, New York (1997)

[Alo90] Alon, N.: Generating pseudo-random permutations and maximum flow algorithms. *Inform. Proc. Letters* **35**, 201–204 (1990)

[Alt88] Althöfer, I.: On optimal realizations of finite metric spaces by graphs. *Discr. Comput. Geom.* **3**, 103–122 (1988)

[And71] Anderson, I.: Perfect matchings of a graph. *J. Comb. Th.* **10**, 183–186 (1971)

[And90] Anderson, I.: *Combinatorial Designs: Construction Methods.* Ellis Horwood Ltd., Chichester (1990)

[And97] Anderson, I.: *Combinatorial Designs and Tournaments.* Oxford University Press, Oxford (1997)

[And77] Anderson, L.D.: On edge-colorings of graphs. *Math. Scand.* **40**, 161–175 (1977)

[AnHa67] Anderson, S.S., Harary, F.: Trees and unicyclic graphs. *Math. Teacher* **60**, 345–348 (1967)

[Ans85] Anstee, R.P.: An algorithmic proof of Tutte's f-factor theorem. *J. Algor.* **6**, 112–131 (1985)

[ApHa77] Appel, K., Haken, W.: Every planar map is 4-colorable. I. Discharging. *Illinois J. Math.* **21**, 429–490 (1977)

[ApHa89] Appel, K., Haken, W.: *Every Planar Map is Four Colorable.* American Mathematical Society, Providence, RI. (1989)

[ApHK77] Appel, K., Haken, W., Koch, J.: Every planar map is 4-colorable: II. Reducibility. *Illinois J. Math.* **21**, 491–567 (1989)

[ApBCC95] Applegate, D., Bixby, R., Chvátal, V., Cook, W.: Finding cuts in the TSP (A preliminary report). DIMACS Technical Report 95-05 (1995)

[ApBCC98] Applegate, D., Bixby, R., Chvátal, V., Cook, W.: On the solution of traveling salesman problems. *Documenta Mathematica* (Extra Volume ICM 1998) **III**, 645–656 (1998).

[ApBCC01] Applegate, D., Bixby, R., Chvátal, V., Cook, W.: TSP cuts which do not conform to the template paradigm. In: Jünger, M., Naddef, D. (eds) *Computational Combinatorial Optimization*, pp. 261–304. Springer, Heidelberg (2001).

[ApBCC03] Applegate, D., Bixby, R., Chvátal, V., Cook, W.: Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems. *Math. Progr.* **97B**, 91–153 (2003)

[ApBCC04] Applegate, D., Bixby, R., Chvátal, V., Cook, W.: Concorde (2004). Available at `www.tsp.gatech.edu`

[ApBCC06] Applegate, D., Bixby, R., Chvátal, V., Cook, W.: *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, Princeton (2006).

[ApCo93] Applegate, D., Cook, W.: Solving large-scale matching problems. In: Johnson, D.S., McGeoch, C.C. (eds) *Network Flows and Matching*, pp. 557–576. Amer. Math. Soc., Providence (1993)

[ArPa86] Arkin, E.M., Papadimitriou, C.H.: On the complexity of circulations. *J. Algor.* **7**, 134–145 (1986)

[Aro98] Arora, S.: Polynomial-time approximation schemes for Euclidean TSP and other geometric problems. *J. Ass. Comp. Mach.* **45**, pp. 753–782 (1998).

[ArLMS92] Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification and hardness of approximation problems. In: *Proc.* 33^{th} *IEEE Symp. on Foundations of Computer Science*, pp. 14–23 (1992)

[ArSa02] Arora, S., Safra, S.: Probabilistic checking of proofs: A new characterization of NP. In: *Proc.* 33^{rd} *IEEE Symp. on Foundations of Computer Science*, pp. 2–13 (1992)

[AuIMN91] Ausiello, G., Italiano, G.F., Marchetti Spaccamela, A., Nanni, U.: Incremental algorithms for minimal length paths. *J. Algor.* **12**, 615–638 (1991)

[Avi78] Avis, D.: Two greedy heuristics for the weighted matching problem. *Congr. Numer.* **21**, 65–76 (1978)

[Avi83] Avis, D.: A survey of heuristics for the weighted matching problem. *Networks* **13**, 475–493 (1983)

[BaFLS91] Babai, L., Fortnow, L., Levin, L.A., and Szegedy, M.: Checking computations in polylogarithmic time. In: *Proc.* 23^{rd} *ACM Symp. on Theory of Computing*, pp. 21–31 (1991)

[BaFL91] Babai, L., Fortnow, L., Lund, C.: Nondeterministic exponential time has two-prover interactive protocols. *Comput. Complexity* **1**, 3–40 (1991)

[Bae53] Bäbler, F.: Über eine spezielle Klasse Eulerscher Graphen. *Comment. Math. Helv.* **21**, 81–100 (1953)

[BaKe92] Bachem, A., Kern, W.: *Linear Programming Duality. An Introduction to Oriented Matroids.* Springer, Berlin (1992)

[Bac89] Bachmann, F.: *Ebene Spiegelungsgeometrie.* B.I. Wissenschaftsverlag, Mannheim-Wien-Zürich (1989)

[BaWi77] Baker, R.D., Wilson, R.M.: Nearly Kirkman triple systems. *Util. Math.* **11**, 289–296 (1977)

[BaFi93] Balas, E., Fischetti, M.: A lifting procedure for the asymmetric traveling salesman polytope and a large new class of facets. *Math. Progr.* **58**, 325–352 (1993)

[BaXu91] Balas, E., Xue, J.: Minimum weighted coloring of triangulated graphs, with application to maximum weight vertex packing and clique finding in arbitrary graphs. *SIAM J. Comp.* **20**, 209–221 (1991):

[BaYu86] Balas, E., Yu, C.S.: Finding a maximum clique in an arbitrary graph. *SIAM J. Comp.* **15**, 1054–1068 (1986)

[BaGo91] Balinsky, M.L., Gonzales, J.: Maximum matchings in bipartite graphs via strong spanning trees. *Networks* **21**, 165–179 (1991)

[BaRa97] Balinski, M., Ratier, G.: On stable marriages and graphs, and strategy and polytopes. *SIAM Review* **39**, 575–604 (1997)

[Bal85] Ball, M.O.: Polynomial algorithms for matching problems with side constraints. Research report CORR 85–21, University of Waterloo (1985)

[BaDe83] Ball, M.O., Derigs, U.: An analysis of alternate strategies for implementing matching algorithms. *Networks* **13**, 517–549 (1983)

[BaCo87] Ball, W.W.R., Coxeter, H.S.M.: *Mathematical Recreations and Essays (13th Edition).* Dover, New York (1987)

[Ban90] Bandelt, H.-J.: Recognition of tree matrices. *SIAM J. Discr. Math.* **3**, 1–6 (1990)

[BaKP93] Bar-Ilan, J., Kortsarz, G., Peleg, D.: How to allocate network centers. *J. Algor.* **15**, 385–415 (1993)

[Bar90] Barahona, F.: On some applications of the chinese postman problem. In: Korte, B., Lovász, L., Prömel, H.J., Schrijver, A. (eds) *Paths, Flows and VLSI-Layout.* Springer, Berlin, pp. 1–16 (1990)

[BaPu87] Barahona, F., Pulleyblank, W.R.: Exact arborescences, matchings and cycles. *Discr. Appl. Math.* **16**, 91–99 (1987)

[BaTa89] Barahona, F., Tardos, E.: Note on Weintraub's minimum-cost circulation algorithm. *SIAM J. Comp.* **18**, 579–583 (1989)

[Bar75] Baranyai, Z.: On the factorization of the complete uniform hypergraph. In: *Proc. Erdős-Koll. Keszthely 1973*, North Holland, Amsterdam, pp. 91–108 (1975)

[BaSa95] Barnes, T.M., Savage, C.D.: A recurrence for counting graphical partitions. *Electronic J. Comb.* **2**, # R 11 (1995)

[BaWo82] Bauer, F.L., Wössner, H.: *Algorithmic Language and Program Development.* Springer, Berlin (1982)

[BaJS90] Bazaraa, M.S., Jarvis, J.J., Sherali, H.D.: *Linear Programming and Network Flows (Second Edition).* Wiley, New York (1990)

[BaSS93] Bazaraa, M.S., Sherali, H.D., Shetty, C.M.: *Nonlinear Programming: Theory and Algorithms (Second Edition).* Wiley, New York (1993)

[Bel58] Bellman, R.E.: On a routing problem. *Quart. Appl. Math.* **16**, 87–90 (1958)

[Ben90] Bentley, J.L.: Experiments on traveling salesman heuristics. In: *Proc. First SIAM Symp. on Discr. Algorithms*, pp. 91–99 (1990)

[Ber57] Berge, C.: Two theorems in graph theory. *Proc. Nat. Acad. Sc. USA* **43**, 842–844 (1957)

[Ber58] Berge, C.: Sur le couplage maximum d'un graphe. *C.R. Acad. Sci. Paris* **247**, 258–259 (1958)

[Ber61] Berge, C.: Färbung von Graphen, deren sämtliche bzw. deren ungerade Kreise starr sind (Zusammenfassung). *Wiss. Z. Martin-Luther-Universität Halle-Wittenberg, Math.-Nat. Reihe* **10**, 114–115 (1961)

[Ber73] Berge, C.: *Graphs and Hypergraphs.* North Holland, Amsterdam (1973)

[BeCh84] Berge, C., Chvátal, V.: *Topics in Perfect Graphs.* North Holland, Amsterdam (1984)

[BeFo91] Berge, C., Fournier, J.C.: A short proof for a generalization of Vizing's theorem. *J. Graph Th.* **15**, 333–336 (1991)

[BeGh62] Berge, C., Ghouila-Houri, A.: *Programmes, Jeux et Réseaux de Transport.* Dunod, Paris (1991)

[BeMT94] Berlekamp, E.R., McEliece, R.J., van Tilborg, H.C.A.: On the inherent intractibility of certain coding problems. *IEEE Trans. Inf. Th.* **24**, 384–386 (1978).

[BeRa94] Berman, P., Ramaiyer, V.: Improved approximations for the Steiner tree problem. *J. Algor.* **17**, 381–408 (1994)

[Ber78] Bermond, J.C.: Hamiltonian graphs. In: Beineke, L.W., Wilson, R.J. (eds) *Selected Topics in Graph Theory*, pp. 127–167. Academic Press, New York (1978)

[Ber92] Bermond, J.C. (ed): *Interconnection Networks.* North Holland, Amsterdam (1992)

[Ber93] Bertsekas, D.P.: A simple and fast label correcting algorithm for shortest paths. *Networks* **23**, 703–709 (1993)

[Bet74] Beth, T.: Algebraische Auflösungsalgorithmen für einige unendliche Familien von 3–Designs. *Le Matematiche* **29**, 105–135 (1974)

[BeJL99] Beth, T., Jungnickel, D., Lenz, H.: *Design Theory (Second Edition, Volumes 1 and 2).* Cambridge University Press, Cambridge (1999)

[Bie89] Bien, F.: Constructions of telephone networks by group representations. *Notices Amer. Math. Soc.* **36**, 5–22 (1989)

[BiBM90] Bienstock, D., Brickell, E.F., Monma, C.N.: On the structure of minimum-weight k-connected spanning networks. *SIAM J. Discr. Math.* **3**, 320–329 (1990)

[Big93] Biggs, N.L.: *Algebraic Graph Theory (Second Edition).* Cambridge University Press, Cambridge (1993)

[BiLW76] Biggs, N.L., Lloyd, E.K., Wilson, R.J.: *Graph Theory 1736–1936.* Oxford University Press, Oxford (1976)

[Bir46] Birkhoff, G.: Tres observaciones sobre el algebra lineal. *Univ. Nac. Tucumán Rev. Ser. A* **5**, 147–151 (1946)

[BjLSW92] Bjørner, A., Las Vergnas, M., Sturmfels, B., White, N., Ziegler, G.M.: *Oriented Matroids.* Cambridge University Press, Cambridge (1992)

[BjZi92] Bjørner, A., Ziegler, G.M.: Introduction to greedoids. In: White, N. (ed) *Matroid Applications*, pp. 284–357. Cambridge University Press, Cambridge (1992)

[Bla83] Blahut, R. E.: *Theory and Practice of Error Control Codes.* Addison-Wesley, Reading, Mass. (1983)

[BlSh89] Bland, R.G., Shallcross, D.F.: Large traveling salesman problems arising from experiments in x-ray crystallography: A preliminary report on computation. *Oper. Res. Letters* **8**, 125–128 (1989)

[BoHa71] Bobrow, L.S., Hakimi, S.L.: Graph theoretic q-ary codes. *IEEE Trans. Inform. Th.* **17**, 215–218 (1971).

[BoTi80] Boesch, F., Tindell, R.: Robbins theorem for mixed multigraphs. *Amer. Math. Monthly* **87**, 716–719 (1980)

[Bol78] Bollobás, B.: *Extremal Graph Theory.* Academic Press, New York (1978)

[BoCh76] Bondy, J.A., Chvátal, V.: A method in graph theory. *Discr. Math.* **15**, 111–135 (1976)

[BoMu76] Bondy, J.A., Murty, U.S.R.: *Graph Theory with Applications.* North Holland, Amsterdam (1976)

[Boo94] Book, R.V.: Relativizations of the P =? NP and other problems: developments in structural complexity theory. *SIAM Review* **36**, 157–175 (1994)

[BoLu76] Booth, S., Lueker, S.: Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. System Sc.* **13**, 335–379 (1976)

[Bor60] Borchardt, C.W.: Über eine der Interpolation entsprechende Darstellung der Eliminationsresultante. *J. Reine Angew. Math.* **57**, 111–121 (1860)

[Bor87] Borgwardt, K.H.: *The Simplex Method. A Probabilistic Analysis.* Springer, Berlin (1987)

[Bor26a] Boruvka, O.: O jistém problému minimálním. *Acta Societ. Scient. Natur. Moravicae* **3**, 37–58 (1926)

[Bor26b] Boruvka, O. : Príspevek k resení otázky ekonomické stavby elektrovodních sítí. *Elektrotechnicky obzor* **15**, 153–154 (1926)

[Bos90] Bosák, J.: *Decompositions of Graphs.* Kluwer Academic Publishers, Dordrecht (1990)

[BoFa90] Boyd, E.A., Faigle, U.: An algorithmic characterization of antimatroids. *Discr. Appl. Math.* **28**, 197–205 (1990)

[BrHa67] Bredeson, J.G., Hakimi, S.L.: Decoding of graph theoretic codes. *IEEE Trans. Inform. Th.* **13**, 348–349 (1967)

[Bro41] Brooks, R.L.: On colouring the nodes of a network. *Proc. Cambridge Phil. Soc.* **37**, 194–197 (1941)

[BrBH90] Brucker, P., Burkard, R.E., Hurink, J.: Cyclic schedules for r irregularly occuring events. *J. Comp. Appl. Math.* **30**, 173–189 (1990)

[BrBr92] Bryant, V., Brooksbank, P.: Greedy algorithm compatibility and heavy-set structures. *Europ. J. Comb.* **13**, 81–86 (1992)

[Bun74] Bunemann, P.: A note on the metric properties of trees. *J. Comb. Th. (B)* **17**, 48–50 (1974)

[Bur86] Burkard, R.E.: Optimal schedules for periodically recurring events. *Discr. Appl. Math.* **15**, 167–180 (1986)

[BuHZ77] Burkard, R.E., Hahn, W., Zimmermann, U.: An algebraic approach to assignment problems. *Math. Progr.* **12**, 318–327 (1977)

[BuGo61] Busacker, R.G., Gowen, P.J.: A procedure for determining a family of minimum cost flow networks. ORO Techn. Report 15, John Hopkins University (1961)

[CaFM79] Camerini, P.M., Fratta, L., Maffioli, F.: A note on finding optimum branchings. *Networks* **9**, 309–312 (1979)

[CaMMT85] Camerini, P.M., Maffioli, F., Martello, S., Toth, P.: Most and least uniform spanning trees. *Discr. Appl. Math.* **15**, 181–197 (1986)

[Cam76] Cameron, P.J.: *Parallelisms of Complete Designs*, Cambridge University Press, Cambridge (1976)

[CaLi91] Cameron, P.J., van Lint, J.H.: *Designs, Graphs, Codes and Their Links.* Cambridge University Press, Cambridge (1991)

[CaRa91] Campbell, D.M., Radford, D.: Tree isomorphism algorithms: Speed versus clarity. *Math. Magazine* **64**, 252–261 (1991)

[CaFT89] Carpaneto, G., Fischetti, M., Toth, P.: New lower bounds for the symmetric travelling salesman problem. *Math. Progr.* **45**, 233–254 (1989)

[Car71] Carré, P.A.: An algebra for network routing problems. *J. Inst. Math. Appl.* **7**, 273–294 (1971)

[Car79] Carré, P.A.: *Graphs and Networks.* Oxford University Press, Oxford (1979)

[Cat79] Catlin, P.A.: Hajós' graph coloring conjecture: variations and counterexamples. *J. Comb. Th. (B)* **26**, 268–274 (1979)

[Cay89] Cayley, A.: A theorem on trees. *Quart. J. Math.* **23**, 376–378 (1889)

[ChAN81] Chandrasekaran, R., Aneja, Y.P., Nair, K.P.K.: Minimal cost reliability ratio spanning tree. *Ann. Discr. Math.* **11**, 53–60 (1981)

[ChTa84] Chandrasekaran, R., Tamir, A.: Polynomial testing of the query 'Is $a^b \geq c^d$' with application to finding a minimal cost reliability ratio spanning tree. *Discr. Appl. Math.* **9**, 117–123 (1984)

[ChGK06] Charikar, M., Hoemans, M.X., Karloff, H.: On the integrality ratio for the asymmetric traveling salesman problem. *Math. Oper. Res.* **31**, 245-252 (2006)

[Cha66] Chartrand, G.: A graph-theoretic approach to communication problems. *SIAM J. Appl. Math.* **14**, 778–781 (1966)

[ChHa68] Chartrand, G., Harary, F.: Graphs with described connectivities. In: Erdős, P., Katona, G. (eds) *Theory of Graphs*, pp. 61–63. Academic Press, New York (1968)

[ChWh70] Chartrand, G., White, A.T.: Randomly transversable graphs. *Elem. Math.* **25**, 101–107 (1970)

[ChHu92] Cheng, C.K., Hu, T.C.: Maximum concurrent flows and minimum cuts. *Algorithmica* **8**, 233–249 (1992)

[ChTa76] Cheriton, D., Tarjan, R.E.: Finding minimum spanning trees. *SIAM J. Comp.* **5**, 724–742 (1976)

[ChHa89] Cheriyan, J., Hagerup, T.: A randomized maximum flow algorithm. In: *Proc.* 30^{th} *IEEE Conf. on Foundations of Computer Science*, pp. 118–123 (1989)

[ChHa95] Cheriyan, J., Hagerup, T.: A randomized maximum flow algorithm. *SIAM J. Computing* **24**, 203–226 (1995)

[ChHM96] Cheriyan, J., Hagerup, T., Mehlhorn, K.: A $O(n^3)$-time maximum flow algorithm. *SIAM J. Computing* **25**, 1144–1170 (1996)

[ChMa89] Cheriyan, J., Maheshwari, S.N.: Analysis of preflow push algorithms for maximum network flow. *SIAM J. Comp.* **18**, 1057–1086 (1989)

[ChGo95] Cherkassky, B.V., Goldberg, A.V.: On implementing Push-Relabel method for the maximum flow problem. In: Balas, E., Clausen, J. (eds) *Integer Programming and Combinatorial Optimization*, pp. 157–171. Springer, Berlin (1995)

[Che80] Cheung, T.Y.: Computational comparison of eight methods for the maximum network flow problem. *ACM Trans. Math. Software* **6**, 1–16 (1980)

[Chr75] Christofides, N.: *Graph Theory: An Algorithmic Approach*. Academic Press, New York (1975)

[Chr76] Christofides, N.: Worst-case analysis of a new heuristic for the travelling salesman problem. Report 388, Grad. School of Ind. Admin., Carnegie-Mellon University (1976)

[ChLi65] Chu, Y.J., Liu, T.H.: On the shortest arborescence of a directed graph. *Sci. Sinica* **14**, 1396–1400 (1965)

[ChRST02] Chudnovsky, M., Robertson, N., Seymour, P.D., Thomas, R.: The strong perfect graph theorem. Preprint (2002)

[ChRST03] Chudnovsky, M., Robertson, N., Seymour, P.D., Thomas, R.: Progress on perfect graphs. *Math. Progr. Ser. B* **97**, 405–422 (2003)

[Chu86] Chung, F.R.K.: Diameters of communication networks. *Proc. Symp. Pure Appl. Math.* **34**, 1–18 (1986)

[ChGT85] Chung, F.R.K., Garey, M.R., Tarjan, R.E.: Strongly connected orientations of mixed multigraphs. *Networks* **15**, 477–484 (1985)

[Chv83] Chvátal, V.: *Linear Programming*. Freeman, New York (1983)

[Chv85] Chvátal, V.: Hamiltonian cycles. In: Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., Shmoys, D.B. (eds) *The Travelling Salesman Problem*, pp. 403–429. Wiley, New York (1985)

[ChEr72] Chvátal, V., Erdős, P.: A note on Hamiltonian circuits. *Discr. Math.* **2**, 111–113 (1972)
[ChTh78] Chvátal, V., Thomasson, C.: Distances in orientations of graphs. *J. Comb. Th. (B)* **24**, 61–75 (1978)
[Cie98] Cieslik, D.: *Steiner Minimal Trees.* Kluwer, Dordrecht (1998)
[Cie01] Cieslik, D.: *The Steiner Ratio.* Kluwer, Dordrecht (2001)
[Col87] Colbourn, C.J.: *The Combinatorics of Network Reliability.* Oxford University Press, Oxford (1987)
[CoDN89] Colbourn, C.J., Day, R.P.J., Nel, L.D.: Unranking and ranking spanning trees of a graph. *J. Algor.* **10**, 271–286 (1989)
[CoHo82] Cole, R., Hopcroft, J.: On edge coloring bipartite graphs. *SIAM J. Comp.* **11**, 540–546 (1982)
[CoHMW92] Conrad, A., Hindrichs, T., Morsy, H., Wegener, I.: Wie es einem Springer gelingt, Schachbretter von beliebiger Größe zwischen beliebig vorgegebenen Anfangs- und Endfeldern vollständig abzureiten. *Spektrum der Wiss.*, pp. 10–14 (Feb. 1992)
[CoHMW94] Conrad, A., Hindrichs, T., Morsy, H., Wegener, I. (1994): Solution of the knight's Hamiltonian path problem on chessboards. *Discr. Appl. Math.* **50**, 125–134 (1992)
[Coo71] Cook, S.A.: The complexity of theorem proving procedures. In: *Proc. 3^{rd} ACM Symp. on the Theory of Computing*, pp. 151–158 (1971)
[CoHa90] Cook, W., Hartmann, M.: On the complexity of branch and cut methods for the traveling salesman problem. In: Cook, W., Seymour, P.D. (eds) *Polyhedral Combinatorics*, pp. 75–81. American Mathematical Society, Providence (1990)
[CoLR90] Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts (1990)
[CoNe78] Cornuejols, G., Nemhauser, G.L.: Tight bounds for Christofides' traveling salesman heuristic. *Math. Progr.* **14**, 116–121 (1978)
[CoRo41] Courant, R., Robbins, H.: *What is Mathematics?* Oxford University Press, New York (1941)
[Cox61] Coxeter, H.M.S.: *Introduction to Geometry.* Wiley, New York (1961)
[Cox73] Coxeter, H.M.S.: *Regular Polytopes (Third Edition)* Dover, New York (1973)
[Cro58] Croes, G.A.: A method for solving traveling-salesman problems. *Oper. Res.* **6**, 791–812 (1958)
[CrPa80] Crowder, H., Padberg, M.W.: Solving large-scale symmetric travelling salesman problems to optimality. *Management Sc.* **26**, 495–509 (1980)
[Cun86] Cunningham, W.H.: Improved bounds for matroid partition and intersection algorithms. *SIAM J. Comp.* **15**, 948–957 (1986)
[CuMa78] Cunningham, W.H., Marsh, A.B.: A primal algorithm for optimal matching. *Math. Progr. Stud.* **8**, 50–72 (1978)
[CvDGT87] Cvetkovic, D.M., Doob, M., Gutman, I., Torgasev, A.: *Recent Results in the Theory of Graph Spectra.* North Holland, New York (1987)
[CvDS80] Cvetkovic, D.M., Doob, M., Sachs, H.: *Spectra of Graphs.* Academic Press, New York (1980)
[DaFF56] Dantzig, G.B., Ford, L.R., Fulkerson, D.R.: A primal-dual algorithm for linear programs. In: Kuhn, H.W., Tucker, A.W. (eds) *Linear Inequalities and Related Systems*, pp. 171–181. Princeton University Press, Princeton (1956)

[DaFJ54] Dantzig, G.B., Fulkerson, D.R., Johnson, S.M.: Solution of a large-scale traveling-salesman problem. *Oper. Res.* **2**, 393–410 (1954)

[deB46] de Bruijn, N.G.: A combinatorial problem. *Indag. Math.* **8**, 461–467 (1946)

[deBA51] de Bruijn, N.G., van Aardenne-Ehrenfest, T.: Circuits and trees in oriented linear graphs. *Simon Stevin* **28**, 203–217 (1951)

[deW80] de Werra, D.: Geography, games and graphs. *Discr. Appl. Math.* **2**, 327–337 (1980)

[deW81] de Werra, D.: Scheduling in sports. *Ann. Discr. Math.* **11**, 381–395 (1981)

[deW82] de Werra, D.: Minimizing irregularities in sports schedules using graph theory. *Discr. Appl. Math.* **4**, 217–226 (1982)

[deW88] de Werra, D.: Some models of graphs for scheduling sports competitions. *Discr. Appl. Math.* **21**, 47–65 (1988)

[deWJM90] de Werra, D., Jacot-Descombes, L., Masson, P.: A constrained sports scheduling problem. *Discr. Appl. Math.* **26**, 41–49 (1990)

[DePK82] Deo, N., Prabhu, G.M., Krishnamoorthy, M.S.: Algorithms for generating fundamental cycles in a graph. *ACM Trans. Math. Software* **8**, 26–42 (1982)

[Der88] Derigs, U.: *Programming in Networks and Graphs.* Springer, Berlin (1988)

[DeHe80] Derigs, U., Heske, A.: A computational study on some methods for solving the cardinality matching problem. *Angew. Inform.* **22**, 249–254 (1980)

[DeMe89] Derigs, U., Meier, W.: Implementing Goldberg's max-flow algorithm - A computational investigation. *ZOR* **33**, 383–403 (1989)

[DeMe91] Derigs, U., Metz, A.: Solving (large scale) matching problems combinatorially. *Math. Progr.* **50**, 113–121 (1991)

[Dij59] Dijkstra, E.W.: A note on two problems in connexion with graphs. *Numer. Math.* **1**, 269–271 (1959)

[Dil50] Dilworth, R.P.: A decomposition theorem for partially ordered sets. *Annals Math.* **51**, 161–166 (1950)

[Din70] Dinic, E.A.: Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.* **11**, 1277–1280 (1970)

[Dir52] Dirac, G.A.: Some theorems on abstract graphs. *Proc. London Math. Soc. (3)* **2**, 69–81 (1952)

[DiRT92] Dixon, B., Rauch, M., Tarjan, R.E.: Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Computing* **21**, 1184–1192 (1992)

[Dom89] Domschke, W.: Schedule synchronization for public transit networks. *OR Spektrum* **11**, 17–24 (1989)

[Dre84] Dress, A.: Trees, tight extensions of metric spaces, and the cohomological dimension of certain groups: A note on combinatorial properties of metric spaces. *Adv. Math.* **53**, 321–402 (1984)

[DuHw90a] Du, D.-Z., Hwang, F.: An approach for proving lower bounds: Solution of Gilbert-Pollak's conjecture on Steiner ratio. In: *Proc. 31st Annual Symp. on Foundations of Computer Science*, Los Alamitos, Cal., pp. 76–85. IEEE Computer Society (1990)

[DuHw90b] Du, D.-Z., Hwang, F.: The Steiner ratio conjecture of Gilbert and Pollak is true. *Proc. National Acad. of Sciences USA* **87**, 9464–9466 (1990)

[DuSR00] Du, D.-Z., Smith, J.M., Rubinstein, J.H.: *Advances in Steiner Trees.* Kluwer, Dordrecht (2000)

[DuZh92] Du, D.-Z., Zhang, Y.: On better heuristics for Steiner minimum trees. *Math. Progr.* **57**, 193–202 (1992)

[Due93] Dueck, G.: New optimization heuristics: The great deluge algorithm and record-to-record travel. *J. Comput. Physics* **104**, 86–92 (1993)

[DuSc90] Dueck, G., Scheuer, T.: Threshold accepting: A general purpose optimization algorithm appearing superior to simulating annealing. *J. Comput. Physics* **90**, 161–175 (1990)

[Eco83] Eco, U.: *The Name of the Rose.* Harcourt Brace Jovanovich, Inc., Martin Secker & Warburg Limited (1983)

[Edm65a] Edmonds, J.: Maximum matching and a polytope with $0,1$-vertices. *J. Res. Nat. Bur. Stand. B* **69**, 125–130 (1965)

[Edm65b] Edmonds, J.: Paths, trees and flowers. *Canad. J. Math.* **17**, 449–467 (1965)

[Edm67a] Edmonds, J.: *An introduction to matching.* Lecture Notes, Univ. of Michigan (1967)

[Edm67b] Edmonds, J.: Optimum branchings. *J. Res. Nat. Bur. Stand. B* **71**, 233–240 (1967)

[Edm70] Edmonds, J.: Submodular functions, matroids and certain polyhedra. In: Guy, K. (ed) *Combinatorial Structures and Their Applications*, pp. 69–87. Gordon & Breach, New York (1970)

[Edm71] Edmonds, J.: Matroids and the greedy algorithm. *Math. Progr.* **1**, 127–136 (1971)

[Edm73] Edmonds, J.: Edge disjoint branchings. In: Rustin, R. (ed) *Combinatorial Algorithms*, pp. 91–96. Algorithmics Press (1973)

[Edm79] Edmonds, J.: Matroid intersection. *Ann. Discr. Math.* **4**, 39–49 (1979)

[EdFu65] Edmonds, J., Fulkerson, D.R.: Transversals and matroid partition. *J. Res. Nat. Bur. Stand. B* **69**, 147–153 (1965)

[EdGi77] Edmonds, J., Giles, R.: A min-max relation for submodular functions on graphs. *Ann. Discr. Math.* **1**, 185–204 (1977)

[EdGi84] Edmonds, J., Giles, R.: Total dual integrality of linear systems. In: Pulleyblank, W.R. (ed) *Progress in Combinatorial Optimization*, pp. 117–129. Academic Press Canada (1984)

[EdJo73] Edmonds, J., Johnson, E.L.: Matching, Euler tours and the Chinese postman. *Math. Progr.* **5**, 88–124 (1973)

[EdKa72] Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. *J. Ass. Comp. Mach.* **19**, 248–264 (1972)

[EdLP82] Edmonds, J., Lovász, L., Pulleyblank, W.R.: Brick decompositions and the matching rank of graphs. *Combinatorica* **2**, 247–274 (1982)

[Ege31] Egerváry, E.: Matrixok kombinatorius tulajdonságairól. *Mat. Fiz. Lapok* **38**, 16–28 (1931)

[Ego81] Egoritsjev, G.E.: Solution of van der Waerden's permanent conjecture. *Adv. Math.* **42**, 299–305 (1981)

[ElFS56] Elias, P., Feinstein, A., Shannon, C.E.: Note on maximum flow through a network. *IRE Trans. Inform. Th.* **IT-12**, 117–119 (1956)

[Eng97] Engel, K.: *Sperner Theory.* Cambridge University Press, Cambridge (1997)

[Epp94] Eppstein, D.: Offline algorithms for dynamic minimum spanning tree problems. *J. Algor.* **17**, 237–250 (1994)

[ErRT80] Erdős, P., Rubin, A.L., Taylor, H.: Choosability in graphs. *Congr. Numer.* **26**, 125–157 (1980)

[ErSz35] Erdős, P., Szekeres, G.: A combinatorial problem in geometry. *Compos. Math.* **2**, 463–470 (1935)

[ErMcC93] Ervolina, T.R., McCormick, S.T.: Two strongly polynomial cut cancelling algorithms for minimum cost network flow. *Discr. Appl. Math.* **46**, 133–165 (1993)

[Etz86] Etzion, T.: An algorithm for constructing m-ary de Bruijn sequences. *J. Algor.* **7**, 331–340 (1986)

[Eul36] Euler, L.: Solutio problematis ad geometriam situs pertinentis. *Comment. Acad. Sci. Imper. Petropol.* **8**, 128–140 (1736)

[Eul52/53] Euler, L.: Demonstratio nonnullorum insignium proprietatum quibus solida hadris planis inclusa sunt praedita. *Novi Comm. Acad. Sci. Petropol.* **4**, 140–160 (1752/53)

[Eul66] Euler, L.: Solution d'une question curieuse qui ne paroit soumise à aucune analyse. *Mémoires de l'Academie Royale des Sciences et Belles Lettres, Année 1759* **15**, 310–337. Berlin (1766)

[Eve73] Even, S.: *Combinatorial Algorithms.* Macmillan, New York (1973)

[Eve77] Even, S.: Algorithm for determining whether the connectivity of a graph is at least k. *SIAM J. Comp.* **6**, 393–396 (1977)

[Eve79] Even, S.: *Graph Algorithms.* Computer Science Press, Rockville, Md. (1979)

[EvGT77] Even, S., Garey, M.R., Tarjan, R.E.: A note on connectivity and circuits in directed graphs. Unpublished manuscript (1977)

[EvIS76] Even, S., Itai, A., Shamir, A.: On the complexity of timetable and multicommodity flow problems. *SIAM J. Comp.* **5**, 691–703 (1976)

[EvTa75] Even, S., Tarjan, R.E.: Network flow and testing graph connectivity. *SIAM J. Comp.* **4**, 507–512 (1975)

[Fai79] Faigle, U.: The greedy algorithm for partially ordered sets. *Discr. Math.* **28**, 153–159 (1979)

[Fai85] Faigle, U.: On ordered languages and the optimization of linear functions by greedy algorithms. *J. Ass. Comp. Mach.* **32**, 861–870 (1985)

[Fal81] Falikman, D.I.: Proof of the van der Waerden conjecture regarding the permanent of a doubly stochastic matrix. *Math. Notes* **29**, 475–479 (1981)

[FaMi60] Farahat, H.K., Mirsky, L.: Permutation endomorphisms and a refinement of a theorem of Birkhoff. *Proc. Cambridge Phil. Soc.* **56**, 322–328 (1960)

[FeMo95] Feder, T., Motwani, R.: Clique partitions, graph compression and speeding up algorithms. *J. Comput. Syst. Sci.* **51**, 261–272 (1995)

[Fie94] Fiechter, C.-N.: A parallel tabu search algorithm for large traveling salesman problems. *Discr. Appl. Math.* **51**, 243–267 (1994)

[FiSe58] Fiedler, M., Sedlacek, J.: O W-basich orientovanych grafu. *Casopis Pest. Mat.* **83**, 214–225 (1958)

[Fis85] Fishburn, P.C.: *Interval Orders and Interval Graphs: A Study of Partially Ordered Sets.* Wiley, New York (1985)

[Fis81] Fisher, M.L.: The Langrangian method for solving integer programming problems. *Management Sc.* **27**, 1–18 (1981)

[Fle83] Fleischner, H.: Eulerian graphs. In: Beineke, L.W., Wilson, R.J. (eds) *Selected Topics in Graph Theory 2*, pp. 17–53. Academic Press, New York. (1983)

[Fle90] Fleischner, H.: *Eulerian Graphs and Related Topics, Part 1, Vol. 1.* North Holland, Amsterdam (1990)

[Fle91] Fleischner, H.: *Eulerian Graphs and Related Topics, Part 1, Vol. 2.* North Holland, Amsterdam (1991)

[Flo62] Floyd, R.W.: Algorithm 97, Shortest path. *Comm. Ass. Comp. Mach.* **5**, 345 (1962)

[For56] Ford, L.R.: *Network Flow Theory.* Rand Corp., Santa Monica, Cal. (1956)

[FoFu56] Ford, L.R., Fulkerson, D.R.: Maximal flow through a network. *Canad. J. Math.* **8**, 399–404 (1956)

[FoFu57] Ford, L.R., Fulkerson, D.R.: A simple algorithm for finding maximal network flows and an application to the Hitchcock problem. *Canad. J. Math.* **9**, 210–218 (1957)

[FoFu58a] Ford, L.R., Fulkerson, D.R.: Constructing maximal dynamic flows from static flows. *Oper. Res.* **6**, 419–433 (1958)

[FoFu58b] Ford, L.R., Fulkerson, D.R.: Network flow and systems of representatives. *Canad J. Math.* **10**, 78–84 (1958)

[FoFu58c] Ford, L.R., Fulkerson, D.R.: A suggested computation for maximum multi-commodity network flows. *Management Sc.* **5**, 97–101 (1958)

[FoFu62] Ford, L.R., Fulkerson, D.R.: *Flows in Networks.* Princeton University Press, Princeton, N.J. (1962)

[FrTa88] Frank, A., Tardos, E.: Generalized polymatroids and submodular flows. *Math. Progr.* **42**, 489–563 (1988)

[Fre85] Frederickson, G.N.: Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comp.* **14**, 781–798 (1985)

[Fre87] Frederickson, G.N.: Fast algorithms for shortest paths in planar graphs. *SIAM J. Comp.* **16**, 1004–1022 (1987)

[FrTa87] Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses on improved network optimization algorithms. *J. Ass. Comp. Mach.* **34**, 596–615 (1987)

[FrWi94] Fredman, M.L., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comp. Syst. Sc.* **48**, 533–551 (1994)

[FrJu99a] Fremuth-Paeger, C., Jungnickel, D.: Balanced network flows I. A unifying framework for design and analysis of matching algorithms. *Networks* **33**, 1–28 (1999)

[FrJu99b] Fremuth-Paeger, C., Jungnickel, D.: Balanced network flows II. Simple augmentation algorithms. *Networks* **33**, 29–41 (1999)

[FrJu99c] Fremuth-Paeger, C., Jungnickel, D.: Balanced network flows III. Strongly polynomial augmentation algorithms. *Networks* **33**, 43–56 (1999)

[FrJu01a] Fremuth-Paeger, C., Jungnickel, D.: Balanced network flows IV. Duality and structure theory. *Networks* **37**, 194–201 (2001)

[FrJu01b] Fremuth-Paeger, C., Jungnickel, D.: Balanced network flows V. Cycle canceling algorithms. *Networks* **37**, 202–209 (2001)

[FrJu01c] Fremuth-Paeger, C., Jungnickel, D.: Balanced network flows VI. Polyhedral descriptions. *Networks* **37**, 210–218 (2001)

[FrJu02] Fremuth-Paeger, C., Jungnickel, D.: Balanced network flows VII. Primal-dual algorithms. *Networks* **39**, 135–142 (2002)

[FrJu03] Fremuth-Paeger, C., Jungnickel, D.: Balanced network flows VIII. A revised theory of phase ordered algorithms and the $O(\sqrt{n}m \log(n^2/m)/\log n)$ bound for the non-bipartite cardinality matching problem. *Networks* **41**, 137–142 (2003)

[Fro12] Frobenius, G.: Über Matrizen aus nicht negativen Elementen. *Sitzungsber. Preuss. Akad. Wiss.* **1912**, 456–477 (1912)

[Fuj86] Fujishige, S.: An $O(m^3 \log n)$ capacity-rounding algorithm for the minimum-cost circulation problem: A dual framework of Tardos' algorithm. *Math. Progr.* **35**, 298–308 (1986)

[Fuj91] Fujishige, S.: *Submodular Functions and Optimization*. North Holland, Amsterdam (1991)

[Ful56] Fulkerson, D.R.: Note on Dilworth's decomposition theorem for partially ordered sets. *Proc. Amer. Math. Soc.* **7**, 701–702 (1956)

[Ful59] Fulkerson, D.R.: Increasing the capacity of a network: The parametric budget problem. *Management Sc.* **5**, 472–483 (1959)

[Ful61] Fulkerson, D.R.: An out-of-kilter method for minimal cost flow problems. *J. SIAM* **9**, 18–27 (1961)

[Gab76] Gabow, H.N.: An efficient implementation of Edmonds' algorithm for maximum matchings on graphs. *J. Ass. Comp. Mach.* **23**, 221–234 (1976)

[Gab90] Gabow, H.N.: Data structures for weighted matching and nearest common ancestors with linking. In: *Proc. First Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 434–443. Soc. Ind. Appl. Math., Philadelphia (1990)

[GaGST86] Gabow, H.N., Galil, Z., Spencer, T., Tarjan, R.E.: Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica* **6**, 109–122 (1986)

[GaKa82] Gabow, H.N., Kariv, O.: Algorithms for edge coloring bipartite graphs and multigraphs. *SIAM J. Comp.* **11**, 117–129 (1982)

[GaTa88] Gabow, H.N., Tarjan, R.E.: Algorithms for two bottleneck optimization problems. *J. Algor.* **9**, 411–417 (1988)

[GaTa89] Gabow, H.N., Tarjan, R.E.: Faster scaling algorithms for network problems. *SIAM J. Comp.* **18**, 1013–1036 (1989)

[GaTa91] Gabow, H.N., Tarjan, R.E.: Faster scaling algorithms for general graph-matching problems. *J. Ass. Comp. Mach.* **38**, 815–853 (1991)

[Gal57] Gale, D.: A theorem on flows in networks. *Pacific J. Math.* **7**, 1073–1082 (1957)

[Gal68] Gale, D.: Optimal assignments in an ordered set: An application of matroid theory. *J. Comb. Th.* **4**, 176–180 (1968

[GaSh62] Gale, D., Shepley, L.S.: College admissions and the stability of marriage. *Amer. Math. Monthly* **69**, 9–15 (1962)

[Gal80] Galil, Z.: Finding the vertex connectivity of graphs. *SIAM J. Comp.* **9**, 197–199 (1980)

[Gal81] Galil, Z.: On the theoretical efficiency of various network flow algorithms. *Theor. Comp. Sc.* **14**, 103–111 (1981)

[GaMG86] Galil, Z., Micali, S., Gabow, H.: On $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. *SIAM J. Comp.* **15**, 120–130 (1986)

[GaSc88] Galil, Z., Schieber, B.(1988): On funding most uniform spanning trees. *Discr. Appl. Math.* **20**, 173–175 (1986)

[GaTa88] Galil, Z., Tardos, E.: An $O(n^2(m + n \log n) \log n)$ min-cost flow algorithm. *J. Ass. Comp. Mach.* **35**, 374–386 (1986)

[Gal64] Gallai, T.: Elementare Relationen bezüglich der Glieder und trennenden Punkte eines Graphen. *Magyar Tud. Akad. Mat. Kutato Int. Kozl.* **9**, 235–236 (1964)

[Gal67] Gallai, T.: Transitiv orientierbare Graphen. *Acta Math. Acad. Sc. Hungar.* **18**, 25–66 (1967)

[GaMi60] Gallai, T., Milgram, A.N.: Verallgemeinerung eines graphentheoretischen Satzes von Redéi. *Acta Sc. Math.* **21**, 181–186 (1960)

[GaGT89] Gallo, G., Grigoriades, M.D., Tarjan, R.E.: A fast parametric maximum flow algorithm and applications. *SIAM J. Comp.* **18**, 30–55 (1989)

[GaPa88] Gallo, G., Pallottino, S.: Shortest path algorithms. *Annals of Operations Research* **13**, 3–79 (1988)

[GaGJ77] Garey, M.R., Graham, R.L., Johnson, D.S.: The complexity of computing Steiner minimal trees. *SIAM J. Appl. Math.* **32**, 835–859 (1977)

[GaJo76] Garey, M.R., Johnson, D.S.: The complexity of near-optimal graph coloring. *J. Ass. Comp. Mach.* **23**, 43–49 (1976)

[GaJo79] Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, New York (1979)

[GaJS76] Garey, M.R., Johnson, D.S., Stockmeyer, L.J.: Some simplified NP-complete graph problems. *Theoret. Comp. Sc.* **1**, 237–267 (1976)

[GaJT76] Garey, M.R., Johnson, D.S., Tarjan, R.E.: The planar Hamiltonian circuit problem is NP-complete. *SIAM J. Comp.* **5**, 704–714 (1976)

[Gas96] Gasparyan, G.S.: Minimal imperfect graphs: A simple approach. *Combinatorica* **16**, 209–212 (1996)

[GhJu90] Ghinelli, D., Jungnickel, D.: The Steinberg module of a graph. *Archiv Math.* **55**, 503–506 (1990)

[Gho62] Ghouila-Houri, A.: Caractérisation des graphes non orientés dont on peut orienter les arêtes de manière à obtenir le graphe d'une relation d'ordre. *C.R. Acad. Sc. Paris* **254**, 1370–1371 (1962)

[GiPo68] Gilbert, E.N., Pollak, H.O.: Steiner minimal trees. *SIAM J. Appl. Math.* **16**, 1–29 (1968)

[GiHo64] Gilmore, P.C., Hoffman, A.J.: A characterization of comparability graphs and of interval graphs. *Canad. J. Math.* **16**, 539–548 (1964)

[GlKP92] Glover, F., Klingman, D., Phillips, N.V.: *Network Models in Optimization and Their Applications in Practice.* Wiley, New York (1992)

[God93] Godsil, C.D.: *Algebraic Combinatorics.* Chapman and Hall, New York (1993)

[Goe88] Goecke, O.: A greedy algorithm for hereditary set systems and a generalization of the Rado-Edmonds characterization of matroids. *Discr. Appl. Math.* **20**, 39–49 (1988)

[GoGT91] Goldberg, A.V., Grigoriadis, M.D., Tarjan, R.E.: Use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Math. Progr.* **50**, 277–290 (1991)

[GoKa96] Goldberg, A., Karzanov, A.V.: Path problems in skew-symmetric graphs. *Combinatorica* **16**, 353–382 (1996)

[GoPT91] Goldberg, A.V., Plotkin S.A., Tardos, E.: Combinatorial algorithms for the generalized circulation problem. *Math. Oper. Res.* **16**, 351–381 (1991)

[GoTT90] Goldberg, A.V., Tardos, E., Tarjan, R.E.: Network flow algorithms. In: Korte, B., Lovász, L., Prömel, H.J., Schrijver, A. (eds) *Paths, Flows and VLSI-layout*, pp. 101–164. Springer, Berlin (1990)

[GoTa88] Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum flow problem. *J. Ass. Comp. Mach.* **35**, 921–940 (1988)

[GoTa89] Goldberg, A.V., Tarjan, R.E.: Finding minimum-cost circulations by canceling negative cycles. *J. Ass. Comp. Mach.* **36**, 873–886 (1989)

[GoTa90] Goldberg, A.V., Tarjan, R.E.: Solving minimum cost-flow problems by successive approximation. *Math. of Oper. Res.* **15**, 430–466 (1990)

[GoGr88] Goldfarb, D., Grigoriadis, M.D.: A computational comparison of the Dinic and network simplex methods for maximum flow. *Ann. Oper. Res.* **13**, 83–123 (1988)

[GoHa90] Goldfarb, D., Hao, J.: A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2m)$ time. *Math. Progr.* **47**, 353–365 (1990)

[GoHa91] Goldfarb, D., Hao, J.: On strongly polynomial variants of the network simplex algorithm for the maximum flow problem. *Oper. Res. Letters* **10**, 383–387 (1991)

[Gol67] Golomb, S.W.: *Shift Register Sequences.* Holden-Day, San Francisco (1967)

[Gol80] Golumbic, M.C.: *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York (1980)

[GoHu61] Gomory, R.E., Hu, T.C.: Multi-terminal network flows. *J. SIAM* **9**, 551–570 (1961)

[GoHu62] Gomory, R.E., Hu, T.C.: An application of generalized linear programming to network flows. *J. SIAM* **10**, 260–283 (1962)

[GoHu64] Gomory, R.E., Hu, T.C.: Synthesis of a communication network. *J. SIAM* **12**, 348–369 (1964)

[GoMi84] Gondran, M., Minoux, N.: *Graphs and Algorithms.* Wiley, New York (1984)

[Gon92] Gonzaga, C.C.: Path-following methods for linear programming. *SIAM Review* **34**, 167–224 (1992)

[GoJa83] Goulden, I.P., Jackson, D.M.: *Combinatorial Enumeration.* Wiley, New York (1983)

[GrHe85] Graham, R.L., Hell.P.: On the history of the minimum spanning tree problem. *Ann. History of Comp.* **7**, 43–57 (1985)

[GrKl78] Greene, R.C., Kleitman, D.J.: Proof techniques in the theory of finite sets. In: Rota, G.C. (ed) *Studies in Combinatorics*, pp. 22–79. Math. Ass. America (1978)

[Gri88] Griggs, J.R.: Saturated chains of subsets and a random walk. *J. Comb. Th. (A)* **47**, 262–283 (1988)

[GrRo96] Griggs, T., Rosa, A.: A tour of European soccer schedules, or testing the popularity of GK_{2n}. *Bull. ICA* **18**, 65–68 (1996)

[GrKa88] Grigoriadis, M.D., Kalantari, B.: A new class of heuristic algorithms for weighted perfect matching. *J. Ass. Comp. Mach.* **35**, 769–776 (1988)

[Gro80] Grötschel, M.: On the symmetric travelling salesman problem: Solution of a 120-city problem. *Math. Progr. Studies* **12**, 61–77 (1980)

[Gro84] Grötschel, M.: Developments in combinatorial optimization. In: Jäger, W., Moser, J., Remmert, R. (eds) *Perspectives in Mathematics: Anniversary of Oberwolfach 1984*, pp. 249–294. Birkhäuser, Basel (1984)

[Gro85] Grötschel, M.: *Operations Research I.* Lecture Notes, Universität Augsburg (1985)

[GrHo85] Grötschel, M., Holland, G.: Solving matching problems with linear programming. *Math. Progr.* **33**, 243–259 (1985)

[GrHo91] Grötschel, M., Holland, O.: Solution of large-scale symmetric travelling salesman problems. *Math. Progr.* **51**, 141–202 (1991)

[GrJR91] Grötschel, M., Jünger, M., Reinelt, G.: Optimal control of plotting and drilling machines: a case study. *ZOR* **35**, 61–84 (1991)

[GrLS84] Grötschel, M., Lovász, L., Schrijver, A.: Polynomial algorithms for perfect graphs. *Ann. Discr. Math.* **21**, 325–356 (1984)

[GrLS93] Grötschel, M., Lovász, L., Schrijver, A.: *Geometric Algorithms and Combinatorial Optimization (Second Edition).* Springer, Berlin (1993)

[Guc96] Guckert, M.: *Anschlußoptimierung in öffentlichen Verkehrsnetzen – Graphentheoretische Grundlagen, objektorientierte Modellierung und Implementierung.* Ph. D. Thesis, Universität Marburg (1996)

[Gul80] Guldan, F.: Maximization of distances of regular polygones on a circle. *Appl. Math.* **25**, 182–195 (1980)

[Gus87] Gusfield, D.: Three fast algorithms for four problems in stable marriage. *SIAM J. Comp.* **16**, 111–128 (1987)

[Gus88] Gusfield, D.: The structure of the stable roommate problem: Efficient representation and enumeration of all stable assignments. *SIAM J. Comp.* **17**, 742–769 (1988)

[Gus90] Gusfield, D.: Very simple methods for all pairs network flow analysis. *SIAM J. Comp.* **19**, 143–155 (1990)

[GuIr89] Gusfield, D., Irving, R.W.: *The Stable Marriage Problem. Structure and Algorithms.* The MIT Press, Cambridge, Mass. (1989)

[GuMF87] Gusfield, D., Martel, C., Fernandez-Baca, D.: Fast algorithms for bipartite network flow. *SIAM J. Comp.* **16**, 237–251 (1987)

[GuNa91] Gusfield, D., Naor, D.: Efficient algorithms for generalized cut-trees. *Networks* **21**, 505–520 (1991)

[GuPa02] Gutin, G., Punnen, A.: *The Traveling Salesman Problem and its Variations.* Kluwer, Dordrecht (2002)

[Had61] Hadley, G.: *Linear Algebra.* Addison-Wesley, Reading, Mass. (1961)

[Had75] Hadlock, F.O.: Finding a maximum cut of a planar graph in polynomial time. *SIAM J. Comp.* **4**, 221–225 (1975)

[Had43] Hadwiger, H.: Über eine Klassifikation der Streckenkomplexe. *Viertelj. Schr. Naturforsch. Ges. Zürich* **88**, 133–142 (1943)

[Haj61] Hajós, G.: Über eine Konstruktion nicht n-färbbarer Graphen. *Wiss. Z. Martin-Luther-Univ. Halle-Wittenberg, Math.-Nat. Reihe* **10**, 116–117 (1961)

[Hak66] Hakimi, S.L.: Recent progress and new problems in applied graph theory. In: *Proc. IEEE Region Six Annual Conf.*, 635–643 (1966)

[HaBr68] Hakimi, S.L., Bredeson, J.G.: Graph theoretic error-correcting codes. *IEEE Trans. Inform. Th.* **14**, 584–591 (1968)

[HaBr69] Hakimi, S.L., Bredeson, J.G.: Ternary graph theoretic error-correcting codes. *IEEE Trans. Inform. Th.* **15**, 435–436 (1969).

[HaFr65] Hakimi, S.L., Frank, H.: Cut set matrices and linear codes. *IEEE Trans. Inform. Th.* **11**, 457–458 (1965)

[HaVa64] Hakimi, S.L., Yau, S.S.: Distance matrix of a graph and its realizability. *Quart. Appl. Math.* **22**, 305–317 (1964)

[Hal56] Hall, M.: An algorithm for distinct representatives. *Amer. Math. Monthly* **63**, 716–717 (1956)

[Hal86] Hall, M.: *Combinatorial Theory (Second Edition).* Wiley, New York (1986)

[Hal35] Hall, P.: On representatives of subsets. *J. London Math. Soc.* **10**, 26–30 (1935)

[HaVa50] Halmos, P.R., Vaughan, H.E.: The marriage problem. *Amer. J. Math.* **72**, 214–215 (1950)

[HaRu94] Hamacher, H.W., Ruhe, G.: On spanning tree problems with multiple objectives. *Ann. Oper. Res.* **52**, 209–230 (1994)

[Har62] Harary, F.: The maximum connectivity of a graph. *Proc. Nat. Acad. Sci. USA* **48**, 1142–1146 (1962)

[Har69] Harary, F.: *Graph Theory.* Addison Wesley, Reading, Mass. (1969)

[HaTu65] Harary, F., Tutte, W.T.: A dual form of Kuratowski's theorem. *Canad. Math. Bull.* **8**, 17–20 and 173 (1965)

[HaJo85] Hassin, R., Johnson, D.B.: An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks. *SIAM J. Comp.* **14**, 612–624 (1985)

[HaKo81] Hausmann, D., Korte, B.: Algorithmic versus axiomatic definitions of matroids. *Math. Progr. Studies* **14**, 98–111 (1981)

[Hea90] Heawood, P.J.: Map colour theorem. *Quart. J. Pure Appl. Math.* **24**, 332–338 (1890)

[HeKa70] Held, M., Karp, R.: The travelling salesman problem and minimum spanning trees. *Oper. Res.* **18**, 1138–1162 (1970)

[HeKa71] Held, M., Karp, R.: The travelling salesman problem and minimum spanning trees II. *Math. Progr.* **1**, 6–25 (1971)

[HeWC74] Held, M., Wolfe, P., Crowder, H.P.: Validation of subgradient optimization. *Math. Progr.* **6**, 62–88 (1974)

[HeMS93] Helman, P., Mont, B.M.E., Shapiro, H.D.: An exact characterization of greedy structures. *SIAM J. Discr. Math.* **6**, 274–283 (1993)

[Hel00] Helsgaun, K.: An effective implementation of the Lin-Kernighan traveling salesman heuristic. *Europ. J. Oper. Res.* **126**, 106–130 (2000)

[Her67] Herz, J.C.: *Cours de Théorie des Graphes.* Faculté des Sciences de Lille (1967)

[Hie73] Hierholzer, C.: Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Math. Ann.* **6**, 30–32 (1873)

[HiJa87] Hilton, A.J.W., Jackson, B.: A note concerning the chromatic index of multigraphs. *J. Graph Th.* **11**, 267–272 (1987)

[Hit41] Hitchcock, F.L.: The distribution of a product from several sources to numerous localities. *J. Math. Phys.* **20**, 224–230 (1941)

[HoLC91] Ho, J.-M., Lee, D.T., Chang, C.-H., Wong, C.K.: Minimum diameter spanning trees and related problems. *SIAM J. Computing* **20**, 987–997 (1991)

[HoNS86] Hochbaum, D.S., Nishizeki, T., Shmoys, D.B.: A better than 'best possible' algorithm to edge color multigraphs. *J. Algor.* **7**, 79–104 (1986)

[Hof60] Hoffman, A.J.: Some recent applications of the theory of linear inequalities to extremal combinatorial analysis. In: Bellman, R.E., Hall, M. (eds) *Combinatorial Analysis*, pp. 113–127. Amer. Math. Soc., Providence (1960)

[Hof74] Hoffman, A.J.: A generalization of max flow-min cut. *Math. Progr.* **6**, 352–359 (1974)

[Hof79] Hoffman, A.J.: The role of unimodularity in applying linear inequalities to combinatorial theorems. *Ann. Discr. Math.* **4**, 73–84 (1979)

[HoKr56] Hoffman, A.J., Kruskal, J.B.: Integral boundary points of convex polyhedra. In: Kuhn, H.W., Tucker, A.W. (eds) *Linear Inequalities and Related Systems*, pp. 233–246. Princeton University Press, Princeton (1956)

[HoKu56] Hoffman, A.J., Kuhn, H.W.: On systems of distinct representatives. *Ann. Math. Studies* **38**, 199–206 (1956)

[HoMa64] Hoffman, A.J., Markowitz, H.M.: A note on shortest path, assignment and transportation problems. *Naval Research Logistics Quarterly* **10**, 375–379 (1963)

[Hol81] Holyer, I.J.: The NP-completeness of edge-coloring. *SIAM J. Comp.* **10**, 718–720 (1981)

[HoKa73] Hopcroft, J., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comp.* **2**, 225–231 (1973)

[HoTa73] Hopcroft, J., Tarjan, R.E.: Dividing a graph into triconnected components. *SIAM J. Comp.* **2**, 135–158 (1973)

[HoUl79] Hopcroft, J., Ullman, J.D.: *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, Reading, Mass. (1979)

[Hor87] Horton, J.D.: A polynomial time algorithm to find the shortest cycle basis of a graph. *SIAM J. Comp.* **16**, 358–366 (1987)

[Hu61] Hu, T.C.: The maximum capacity route problem. *Oper. Res.* **9**, 898–900 (1961)

[Hu74] Hu, T.C.: Optimum communication spanning trees. *SIAM J. Comp.* **3**, 188–195 (1974)

[HuMR82] Huang, C., Mendelsohn, E., Rosa, A.: On partially resolvable t-partitions. *Ann. Discr. Math.* **12**, 160–183 (1982)

[HuDi88] Hung, M.S., Divoky, J.J.: A computational study of efficient shortest path algorithms. *Computers and Operations Research* **15**, 567–576 (1988)

[Hup67] Huppert, B.: *Endliche Gruppen I.* Springer, Berlin–Heidelberg (1967)

[HwRW92] Hwang, F.K., Richards, D.S., Winter, P.: *The Steiner Tree Problem.* North Holland, Amsterdam (1992)

[Ima83] Imai, H.: On the practical efficiency of various maximum flow algorithms. *J. Oper. Res. Soc. of Japan* **26**, 61–82 (1983)

[ImSZ84] Imrich, W., Simões-Pereira, J.M.S., Zamfirescu, C.M.: On optimal embeddings of metrics in graphs. *J. Comb. Th. (B)* **36**, 1–15 (1984)

[Irv85] Irving, R.W.: An efficient algorithm for the 'stable roommates' problem. *J. Algor.* **6**, 577–595 (1985)

[IrLe86] Irving, R.W., Leather, P.: The complexity of counting stable marriages. *SIAM J. Comp.* **15**, 655–667 (1986)

[IrLG87] Irving, R.W., Leather, P., Gusfield, D.: An efficient algorithm for the 'optimal' stable marriage. *J. Ass. Comp. Mach.* **34**, 532–543 (1987)

[ItPS82] Itai, A., Perl, Y., Shiloach, Y.: The complexity of finding maximum disjoint paths with length constraints. *Networks* **12**, 277–286 (1982)

[ItRo78] Itai, A., Rodeh, M.: Finding a minimum circuit in a graph. *SIAM J. Comp.* **7**, 413–423 (1978)

[ItRT78] Itai, A., Rodeh, M., Tanimota, S.L.: Some matching problems in bipartite graphs. *J. Ass. Comp. Mach.* **25**, 517–525 (1978)

[ItSh79] Itai, A., Shiloach, Y.: Maximum flow in planar networks. *SIAM J. Comp.* **8**, 135–150 (1979)

[Jar30] Jarník, V.: O jistém problému minimálním. *Acta Societ. Scient. Natur. Moravicae* **6**, 57–63 (1930)

[JaKR93] Jarrah, A.I.Z., Yu, G., Krishnamurthy, N., Rakshit, A.: A decision support framework for airline flight cancellations and delays. *Transportation Sc.* **27**, 266–280 (1993)

[Jel03] Jelliss, G: *Knight's Tour Notes.* `http://www.ktn.freeuk.com/index.htm` (2003)

[Jen76] Jenkyns, T.A.: The efficacy of the 'greedy' algorithm. In: *Proc. 7th Southeastern Conf. Combinatorics, Graph Theory and Computing*, pp. 341–350 (1976)

[JeTo95] Jensen, T.R., Toft, B.: *Graph Coloring Problems.* Wiley, New York (1995)

[JeWi85] Jensen, K., Wirth, N.: *PASCAL User Manual and Report (Third edition).* Springer, New York (1985)

[Joh75] Johnson, D.B.: Priority queues with update and minimum spanning trees. *Inf. Proc. Letters* **4**, 53–57 (1975)

[JoLR78] Johnson, D.S., Lenstra, J.K., Rinnooy Kan, A.H.G.: The complexity of the network design problem. *Networks* **8**, 279–285 (1978)

[JoMcG93] Johnson, D.S., McGeoch, C.C. (eds): *Network Flows and Matching.* American Mathematical Society, Providence (1993)

[JoVe82] Johnson, D.S., Venkatesan, S.M.: Using divide and conquer to find flows in directed planar networks in $O(n^{3/2} \log n)$ time. In: *Proc. 20th Allerton Conf. on Communication, Control and Computing*, pp. 898–905. Univ. of Illinois, Urbana, Ill. (1982)

[Jun79a] Jungnickel, D.: A construction of group divisible designs. *J. Stat. Planning Inf.* **3**, 273–278 (1979)

[Jun79b] Jungnickel, D.: Die Methode der Hilfsmatrizen. In: Tölke, J., Wills, J.M. (eds) *Contributions to Geometry*, pp. 388–394. Birkhäuser, Basel (1979)

[Jun86] Jungnickel, D.: Transversaltheorie: Ein Überblick. *Bayreuther Math. Schriften* **21**, 122–155 (1986)

[Jun93] Jungnickel, D.: *Finite Fields.* B.I. Wissenschaftsverlag, Mannheim (1993)

[JuLe88] Jungnickel, D., Leclerc, M.: A class of lattices. *Ars Comb.* **26**, 243–248 (1988)

[JuLe89] Jungnickel, D., Leclerc, M.: The 2–matching lattice of a graph. *J. Comb. Th. (B)* **46**, 246–248 (1989)

[JuLe87] Jungnickel, D., Lenz, H.: Minimal linear spaces. *J. Comb. Th. (A)* **44**, 229–240 (1987)

[JuVa95] Jungnickel, D., Vanstone, S.A.: An application of coding theory to a problem in graphical enumeration. *Archiv Math.* **65**, 461–464 (1995)

[JuVa96] Jungnickel, D., Vanstone, S.A.: Graphical codes – a tutorial. *Bull. ICA* **18**, 45–64 (1996).

[JuVa97] Jungnickel, D., Vanstone, S.A.: Graphical codes revisited. *IEEE Trans. Inform. Th* **43**, 136–146 (1997).

[JuVa99a] Jungnickel, D., Vanstone, S.A.: Ternary graphical codes. *J. Comb. Math. Comb. Comp.* **29**, 17–31 (1999).

[JuVa99b] Jungnickel, D., Vanstone, S.A.: q-ary graphical codes. *Discr. Math* **208/209**, 375–386 (1999).

[Kah62] Kahn, A.B.: Topological sorting of large networks. *Comm. Ass. Comp. Mach.* **5**, 558–562 (1962)

[Kal60] Kalaba, R.: On some communication network problems. In: Bellman, R.E., Hall, M. (eds) *Combinatorial Analysis*, pp. 261–280. Amer. Math. Soc., Providence (1960)

[Kar99] Karger, D.R.: Using random sampling to find maximum flows in uncapacitated undirected graphs. In: *Proc. 29th. ACM Symp. on Theory of Computing*, pp. 240–249. Ass. Comp. Mach., New York (1999)

[Kar84] Karmarkar, N.: A new polynomial-time algorithm for linear programming. *Combinatorica* **4**, 373–396 (1984)

[Kar72] Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds) *Complexity of Computer Computations*. Plenum Press, New York, pp. 85–103 (1972)

[Kar78] Karp, R.M.: A characterization of the minimum cycle mean in a digraph. *Discr. Math.* **23**, 309–311 (1978)

[Kar74] Karzanov, A.V.: Determining the maximal flow in a network with the method of preflows. *Soviet Math. Dokl.* **15**, 434–437 (1974)

[Kas67] Kasteleyn, P.W.: Graph theory and crystal physics. In: Harary, F. (ed) *Graph Theory and Theoretical Physics*, pp. 43–110. Academic Press, New York (1967)

[KaCh65] Kay, D.C., Chartrand, G.: A characterization of certain ptolemaic graphs. *Canad. J. Math.* **17**, 342–346 (1965)

[Kem79] Kempe, A.B.: On the geographical problem of the four colours. *Amer. J. Math.* **2**, 193–200 (1879)

[Kha79] Khachiyan, L.G.: A polynomial algorithm in linear programming. *Soviet Math. Dokl.* **20**, 191–194 (1979)

[KiRT94] King, V., Rao, S., Tarjan, R.: A faster deterministic maximum flow algorithm. *J. Algorithms* **17**, 447–474 (1994)

[Kirh47] Kirchhoff, G.: Über die Auflösungen der Gleichungen, auf die man bei der Untersuchung der Verteilung galvanischer Ströme geführt wird. *Ann. Phys. Chem.* **72**, 497–508 (1847)

[Kir47] Kirkman, T.P.: On a problem in combinatorics. *Cambridge and Dublin Math. J.* **2**, 191–204 (1847)

[Kir50] Kirkman, T.P.: Query VI. *Lady's and gentleman's diary* **147**, 48 (1850)
[Kle67] Klein, M.: A primal method for minimal cost flows, with applications to the assignment and transportation problems. *Management Sc.* **14**, 205–220 (1967)
[KlWe91] Kleitman, D.J., West, D.B.: Spanning trees with many leaves. *SIAM J. Discr. Math.* **4**, 99–106 (1991)
[KlPh86] Klingman, D., Philipps, N.V.: Network algorithms and applications. *Discr. Appl. Math.* **13**, 107–292 (1986)
[Knu67] Knuth, D.E.: Oriented subtrees of an arc digraph. *J. Comb. Th.* **3**, 309–314 (1967)
[Knu81] Knuth, D.E.: A permanent inequality. *Amer. Math. Monthly* **88**, 731–740 (1981)
[KoSt93] Kocay, W., Stone, D.: Balanced network flows. *Bull. ICA* **7**, 17–32 (1993)
[KoSt95] Kocay, W., Stone, D.: An algorithm for balanced flows. *J. Comb. Math. Comb. Comp.* **19**, 3–31 (1995)
[Koe16] König, D.: Über Graphen und ihre Anwendungen auf Determinantentheorie und Mengenlehre. *Math. Ann.* **77**, 453–465 (1916)
[Koe31] König, D.: Graphen und Matrizen (Hungarian with a summary in German). *Mat. Fiz. Lapok* **38**, 116–119 (1931)
[KoHa78] Korte, B., Hausmann, D.: An analysis of the greedy heuristic for independence systems. *Ann. Discr. Math.* **2**, 65–74 (1978)
[KoLo81] Korte, B., Lovász, L.: Mathematical structures underlying greedy algorithms. In: Gécseg, F. (ed) *Fundamentals of Computation Theory*, pp. 205–209. Springer, Berlin (1981)
[KoLo84] Korte and Lovász (1984) Korte, B., Lovász, L.: Greedoids and linear objective functions. *SIAM J. Algebr. Discr. Math.* **5**, 229–238 (1984)
[KoLP90] Korte, B., Lovász, L., Prömel, H.J., Schrijver, A. (eds): *Paths, Flows and VLSI-Layout.* Springer, Berlin (1990)
[KoLS91] Korte, B., Lovász, L., Schrader, R.: *Greedoids.* Springer, Berlin (1991)
[KoPS90] Korte, B., Prömel, H.J., Steger, A.: Steiner trees in VLSI-Layout. In: Korte, B., Lovász, L., Prömel, H.J., Schrijver, A. (eds) *Paths, Flows and VLSI-layout*, pp. 185–214. Springer, Berlin (1990)
[KoMo89] Korte, N., Möhring, R.H.: An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. Comp.* **18**, 68–81 (1989)
[Kra07] Kravitz, D.: Two comments on minimum spanning trees. *Bull. ICA* **49**, 7–10 (2007)
[Kri75] Krishnamoorthy, M.S.: An NP-hard problem in bipartite graphs. *SIGACT News* **7:1**, 26 (1975)
[Kru56] Kruskal, J.B.: On the shortest spanning subtree of a graph and the travelling salesman problem. *Proc. Amer. Math. Soc.* **7**, 48–50 (1956)
[Kuh55] Kuhn, H.W.: The hungarian method for the assignment problem. *Naval Res. Logistic Quart.* **2**, 83–97 (1955)
[Kuh56] Kuhn, H.W.: Variants of the hungarian method for the assignment problem. *Naval Res. Logistic Quart.* **3**, 253–258 (1956)
[KuSa86] Kuich, W., Salomaa, A.: *Semirings, Automata, Languages.* Springer, Berlin (1986)
[Kur30] Kuratowski, K.: Sur le problème des courbes gauches en topologie. *Fund. Math.* **15**, 271–283 (1930)

[Kwa62] Kwan, M.-K.: Graphic programming using odd and even points. *Chines. Math.* **1**, 273–277 (1962)

[Lam87] Lamken, E.: A note on partitioned balanced tournament designs. *Ars Comb.* **24**, 5–16 (1987)

[LaVa87] Lamken, E., Vanstone, S.A.: The existence of partitioned balanced tournament designs. *Ann. Discr. Math.* **34**, 339–352 (1987)

[LaVa89] Lamken, E., Vanstone, S.A.: Balanced tournament designs and related topics. *Discr. Math.* **77**, 159–176 (1989)

[Las72] Las Vergnas, M.: *Problèmes de couplages et problèmes hamiltoniens en théorie des graphes.* Dissertation, Universitè de Paris VI (1972)

[Law75] Lawler, E.L.: Matroid intersection algorithms. *Math. Progr.* **9**, 31–56 (1975)

[Law76] Lawler, E.L.: *Combinatorial Optimization: Networks and Matriods.* Holt, Rinehart and Winston, New York (1976)

[LaLRS85] Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., Shmoys, D.B. (eds): *The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, New York (1985)

[LaLRS93] Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., Shmoys, D.B.: Sequencing and scheduling: Algorithms and complexity. In: Graves, S.C., Rinnooy Kan, A.H.G., Zipkin, P.H. (eds) *Logistics of Production and Inventory*, pp. 445–522. Elsevier, Amsterdam (1993)

[Lec86] Leclerc, M.: *Polynomial time algorithms for exact matching problems.* M. Math. thesis, University of Waterloo, Dept. of Combinatorics and Optimization (1986)

[Lec87] Leclerc, M.: *Algorithmen für kombinatorische Optimierungsprobleme mit Partitionsbeschränkungen.* Dissertation, Universität Köln (1987)

[LeRe89] Leclerc, M., Rendl, F.: Constrained spanning trees and the travelling salesman problem. *Europ. J. Oper. Res.* **39**, 96–102 (1989)

[Leh64] Lehman, A.: A solution of the Shannon switching game. *SIAM J. Appl. Math.* **12**, 687–725 (1964)

[Len90] Lengauer, T.: *Combinatorial Algorithms for Integrated Circuit Layout.* Wiley, New York (1990)

[LeRi75] Lenstra, J.K., Rinnooy Kan, A.H.G.: Some simple applications of the travelling salesman problem. *Oper. Res. Quart.* **26**, 717–733 (1975)

[LePP84] Lesk, M., Plummer, M.D., Pulleyblank, W.R.: Equi-matchable graphs. In: Bollobás, B. (ed) *Graph Theory and Combinatorics*, pp. 239–254. Academic Press, New York (1984)

[LeOe86] Lesniak, L., Oellermann, O.R.: An Eulerian exposition. *J. Graph Th.* **10**, 277–297 (1986)

[LeLL86] Lewandowski, J.L., Liu, C.L., Liu, J.W.S.: An algorithmic proof of a generalization of the Birkhoff-Von Neumann theorem. *J. Algor.* **7**, 323–330 (1986)

[LePa81] Lewis, H.R., Papadimitriou, C.H.: *Elements of the Theory of Computation.* Prentice Hall, Englewood Cliffs, N. J. (1981)

[LiNi94] Lidl, R., Niederreiter, H.: *Introduction to Finite Fields and Their Applications (Revised Edition).* Cambridge University Press, Cambridge (1994).

[Lin65] Lin, S.: Computer solutions of the travelling salesman problem. *Bell Systems Techn. J.* **44**, 2245–2269 (1965)

[LiKe73] Lin, S., Kernighan, B.W.: An effective heuristic algorithm for the travelling salesman problem. *Oper. Res.* **31**, 498–516 (1973)

[LiLW88] Linial, N., Lovász, L., Widgerson, A.: Rubber bands, convex embeddings and graph connectivity. *Combinatorica* **8**, 91–102 (1988)

[LiMSK63] Little, J.D.C., Murty, K.G., Sweeney, D.W., Karel, C.: An algorithm for the travelling salesman problem. *Oper. Res.* **11**, 972–989 (1963)

[Lom85] Lomonosov, M.V.: Combinatorial approaches to multiflow problems. *Discr. Appl. Math.* **11**, 1–93 (1985)

[Lov70a] Lovász, L.: Problem 11. In: Guy, R., Hanani, H., Sauer, N., Schönheim, J. (eds) *Combinatorial Structures and Their Applications*, pp. 497. Gordon and Breach, New York (1970)

[Lov70b] Lovász, L.: Subgraphs with prescribed valencies. *J. Comb. Th.* **8**, 391–416 (1970)

[Lov72] Lovász, L.: Normal hypergraphs and the perfect graph conjecture. *Discr. Math.* **2**, 253–267 (1972)

[Lov76] Lovász, L.: On two minimax theorems in graph theory. *J. Comb. Th. (B)* **21**, 96–103 (1976)

[Lov79] Lovász, L.: Graph theory and integer programming. *Ann. Discr. Math.* **4**, 141–158 (1979)

[Lov85] Lovász, L.: Some algorithmic problems on lattices. In: Lovász, L., Smerédi, E. (eds) *Theory of Algorithms*, pp. 323–337. North Holland, Amsterdam (1985)

[Lov87] Lovász, L.: The matching structure and the matching lattice. *J. Comb. Th. (B)* **43**, 187–222 (1987)

[LoPl86] Lovász, L., Plummer, M.D.: *Matching Theory.* North Holland, Amsterdam (1986)

[Luc82] Lucas, E.: *Récréations Mathématiques.* Paris (1882)

[Lue82] Lüneburg, H.: Programmbeispiele aus Algebra, Zahlentheorie und Kombinatorik. Report, Universität Kaiserslautern (1982)

[Lue89] Lüneburg, H.: *Tools and Fundamental Constructions of Combinatorial Mathematics.* Bibliographisches Institut, Mannheim (1989)

[Ma94] Ma, S.L.: A survey of partial difference sets. *Designs, Codes and Cryptography* **4**, 221–261 (1994)

[Mac87] Maculan, N.: The Steiner problem in graphs. *Ann. Discr. Math.* **31**, 185–212 (1987)

[MacSl77] MacWilliams, F.J., Sloane, N.J.A.: *The Theory of Error-Correcting Codes.* North Holland, Amsterdam (1977)

[Mad79] Mader, W.: Connectivity and edge-connectivity in finite graphs. In: Bollobás. B. (ed) *Surveys in Combinatorics*, pp. 66–95. Cambridge University Press, Cambridge (1979)

[MaWo94] Magnanti, T.L., Wong, R.T.: Network design and transportation planning: models and algorithms. *Transportation Sci.* **18**, 1–55 (1984)

[MaKM78] Malhotra, V.M., Kumar, M.P., Mahaswari, S.N.: An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Inform. Proc. Letters.* **7**, 277–278 (1978)

[MaSc89] Mansour, Y., Schieber, B.: Finding the edge connectivity of directed graphs. *J. Algor.* **10**, 76–85 (1989)

[MaMi65] Marcus, M., Minc, H.: Diagonal products in doubly stochastic matrices. *Quart. J. Math. (2)* **16**, 32–34 (1965)

[MaRe59] Marcus, M., Ree, R.: Diagonals of doubly stochastic matrices. *Quart. J. Math. (2)* **10**, 296–302 (1959)

[Mar92] Martin, A.: *Packen von Steinerbäumen: Polyedrische Studien und Anwendung.* Dissertation, Technische Universität Berlin (1992)

[MaOF91] Martin, O., Otto, S.W., Felten, E.W.: Large-step Markov chains for the traveling salesman problem. *Complex Systems* **5**, 299–326 (1991)

[Mat95] Matsui, T.: The minimum spanning tree problem on a planar graph. *Discr. Appl. Math.* **58**, 91–94 (1995)

[Mat87] Matula, D.W.: Determining edge connectivity in $O(mn)$. In: *Proc. 28^{th} Symp. on Foundations of Computer Science*, pp. 249–251 (1987)

[McEl87] McEliece, R.J.: *Finite Fields for Computer Scientists and Engineers.* Kluwer, Boston (1987).

[Meh84] Mehlhorn, K.: *Data Structures and Algorithms.* Springer, Berlin (1984)

[MeSc86] Mehlhorn, K., Schmidt, B.H.: On BF-orderable graphs. *Discr. Appl. Math.* **15**, 315–327 (1986)

[MeRo85] Mendelsohn, E., Rosa, A.: One-factorizations of the complete graph – a survey. *J. Graph Th.* **9**, 43–65 (1985)

[MeDu58] Mendelsohn, N.S., Dulmage, A.L.: Some generalizations of the problem of distinct representatives. *Canad. J. Math.* **10**, 230–241 (1958)

[Men74] Meng, D.H.C.: *Matchings and Coverings for Graphs.* Ph.D. thesis, Michigan State University, East Lansing, Mich. (1974)

[Men27] Menger, K.: Zur allgemeinen Kurventheorie. *Fund. Math.* **10**, 96–115 (1927)

[MiVa80] Micali, S., Vazirani, V.V.: An $O(\sqrt{|V||E|})$ algorithm for finding maximum matchings in general graphs. In: *Proc. 21st IEEE Symp. on Foundations of Computer Science*, pp.17–27 (1980)

[Mic92] Michalewicz, Z.: *Genetic Algorithms + Data Structures = Evolution Programs.* Springer, Berlin (1992)

[Mil10] Miller, G.A.: On a method due to Galois. *Quart. J. Pure Appl. Math.* **41**, 382–384 (1910)

[Min78] Minc, H.: *Permanents.* Addison-Wesley, Reading, Mass. (1978)

[Min88] Minc, H.: *Nonnegative Matrices.* Wiley, New York (1988)

[Min60] Minty, G.J.: Monotone networks. *Proc. Royal Soc. London (A)* **257**, 194–212 (1960)

[Min66] Minty, G.J.: On the axiomatic foundations of the theories of directed linear graphs, electrical networks and network programming. *J. Math. Mech.* **15**, 485–520 (1966)

[MiRo84] Mirkin, B.G., Rodin, N.S.: *Genes and Graphs.* Springer, New York (1984)

[Mir69a] Mirsky, L.: Hall's criterion as a 'self-refining' result. *Monatsh. Math.* **73**, 139–146 (1969)

[Mir69b] Mirsky, L.: Transversal theory and the study of abstract independence. *J. Math. Anal. Appl.* **25**, 209–217 (1969)

[Mir71a] Mirsky, L.: A dual of Dilworth's decomposition theorem. *Amer. Math. Monthly* **78**, 876–877 (1971)

[Mir71b] Mirsky, L.: *Transversal Theory.* Academic Press, New York (1971)

[MiPe67] Mirsky, L., Perfect, H.: Applications of the notion of independence to problems of combinatorial analysis. *J. Comb. Th.* **2**, 327–357 (1967)

[Mit99] Mitchell, J.S.B.: Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for Euclidean TSP, k-MST, and related problems. *SIAM J. Comp.* **28**, 1298–1309 (1999)

[MoPo93] Mohar, B., Poljak, S.: Eigenvalues in Combinatorial Optimization. In: Brualdi, R., Friedland, S., Klee, V. (eds) *Combinatorial and Graph-Theoretic Problems in Linear Algebra*, pp. 107–151. Springer, New York (1993)

[MoTh01] Mohar, B., Thomassen, C.: *Graphs on Surfaces.* John Hopkins University Press, Baltimore, Md. (2001)

[Mon83] Monien, B.: The complexity of determining a shortest cycle of even length. *Computing* **31**, 355–369 (1983)

[Moo59] Moore, E.F.: The shortest path through a maze. In: *Proc. Int. Symp. on Theory of Switching Part II*, pp. 285–292. Harvard University Press, Cambridge, Mass. (1959)

[MuGK81] Mühlenbein, H., Gorges-Schleuter, M., Krämer, O.: Evolution algorithms in combinatorial optimization. *Parallel Computing* **7**, 65–85 (1988)

[Mui03] Muirhead, A.F.: Some methods applicable to identities and inequalities of symmetric algebraic functions of n letters. *Proc. Edinburgh Math. Soc.* **21**, 144–157 (1903)

[Mue66] Müller-Merbach, H.: Die Anwendung des Gozinto-Graphs zur Berechnung des Roh- und Zwischenproduktbedarfs in chemischen Betrieben. *Ablauf- und Planungsforschung* **7**, 189–198 (1966)

[Mue69] Müller-Merbach, H.: Die Inversion von Gozinto-Matrizen mit einem graphen-orientierten Verfahren. *Elektron. Datenverarb.* **11**, 310–314 (1969)

[Mue73] Müller-Merbach, H.: *Operations Research (Third Edition).* Franz Vahlen, München (1973)

[Nad90] Naddef, D.: Handles and teeth in the symmetric travelling salesman polytope. In: Cook, W., Seymour, P.D. (eds) *Polyhedral Combinatorics*, pp. 61–74. Amer. Math. Soc., Providence (1990)

[NeWo88] Nemhauser, G.L., Wolsey, L.A.: *Integer and Combinatorial Optimization.* Wiley, New York (1988)

[NiWi78] Nijenhuis, A., Wilf, H.S.: *Combinatorial Algorithms (Second Edition).* Academic Press, New York (1978)

[NiCh88] Nishizeki, T., Chiba, N.: *Planar Graphs: Theory and Algorithms.* North Holland, Amsterdam (1988)

[NoPi96] Nobert, Y., Picard, J.-C.: An optimal algorithm for the mixed Chinese Postman Problem. *Networks* **27**, 95–108 (1996)

[NtHa81] Ntafos, S.C., Hakimi, S.L.: On the complexity of some coding problems. *IEEE Trans. Inform. Th.* **27**, 794–796 (1981)

[Oel96] Oellermann, O.R.: Connectivity and edge-connectivity in graphs: a survey. *Congr. Numer.* **116**, 231–252 (1996)

[Or76] Or, I.: *Traveling salesman-type combinatorial problems and their relation to the logistics of regional blood banking.* Ph.D. thesis, Northwestern University, Evanston, Ill. (1976)

[Ore51] Ore, O.: A problem regarding the tracing of graphs. *Elem. Math.* **6**, 49–53 (1951)

[Ore55] Ore, O.: Graphs and matching theorems. *Duke Math. J.* **22**, 625–639 (1955)

[Ore60] Ore, O.: Note on hamiltonian circuits. *Amer. Math. Monthly* **67**, 55 (1960)

[Ore61] Ore, O.: Arc coverings of graphs. *Ann. Mat. Pura Appl.* **55**, 315–322 (1961)

[Orl93] Orlin, J.B.: A faster strongly polynomial minimum cost flow algorithm. *Oper. Res.* **41**, 338–350 (1993)

[Orl97] Orlin, J.B.: A polynomial time primal network simplex algorithm for minimum cost flows. *Math. Progr.* **78**, 109–129 (1997)

[OrAh92] Orlin, J.B., Ahuja, R.K.: New scaling algorithms for the assignment and minimum cycle mean problems. *Math. Progr.* **54**, 41–56 (1992)

[OrPT93] Orlin, J.B., Plotkin, S.A., Tardos, E.: Polynomial dual network simplex algorithms. *Math. Progr.* **60**, 255–276 (1993)

[Ott48] Otter, R.: The number of trees. *Ann. Math.* **49**, 583–599 (1948)

[Oxl92] Oxley, J.G.: *Matroid Theory.* Oxford University Press, Oxford (1992)

[Oza25] Ozanam, J : *Récréations Mathématiques et Physiques Vol. 1.* Claude Jombert, Paris (1725)

[PaSu91] Padberg, M.W., Sung, T.-Y.: An analytical comparison of different formulations of the travelling salesman problem. *Math. Progr.* **52**, 315–357 (1991)

[PaHo80] Padberg, M.W., Hong, S.: On the symmetric travelling salesman problem: A computational study. *Math. Progr. Studies* **12**, 78–107 (1980)

[PaRa74] Padberg, M.W., Rao, M.R.: The travelling salesman problem and a class of polyhedra of diameter two. *Math. Progr.* **7**, 32–45 (1974)

[PaRi87] Padberg, M.W., Rinaldi, G.: Optimization of a 532-city symmetric travelling salesman problem. *Oper. Res. Letters* **6**, 1–7 (1987)

[PaRi91] Padberg, M.W., Rinaldi, G.: A branch-and-cut algorithm for the resolution of large-scale travelling salesman problems. *SIAM Rev.* **33**, 60–100 (1991)

[Pap76] Papadimitriou, C.H.: On the complexity of edge traversing. *J. Ass. Comp. Mach.* **23**, 544–554 (1976)

[Pap78] Papadimitriou, C.H.: The adjacency relation on the traveling salesman polytope is NP-complete. *Math. Progr.* **14**, 312–324 (1978)

[Pap92] Papadimitriou, C.H.: The complexity of the Lin-Kernighan heuristic for the traveling salesman problem. *SIAM J. Comp.* **21**, 450–465.

[Pap94] Papadimitriou, C.H.: *Computational Complexity.* Addison-Wesley, Reading, Mass. (1994)

[PaSt77] Papadimitriou, C.H., Steiglitz, K.: On the complexity of local search for the travelling salesman problem. *SIAM J. Comp.* **6**, 76–83 (1977)

[PaSt78] Papadimitriou, C.H., Steiglitz, K.: Some examples of difficult travelling salesman problems. *Oper. Res.* **26**, 434–443 (1978)

[PaSt82] Papadimitriou, C.H., Steiglitz, K.: *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall, Englewood Cliffs, N. J. (1982)

[PaVe06] Papadimitriou, C.H., Vempala, S.: On the approximability of the traveling salesman problem. *Combinatorica* **26**, 101–120 (2006)

[PaYa82] Papadimitriou, C.H., Yannakakis, M.: The complexity of restricted spanning tree problems. *J. Ass. Comp. Mach.* **29**, 285–309 (1982)

[PaYa93] Papadimitriou, C.H., Yannakakis, M.: The traveling salesman problem with distances 1 and 2. *Math. of Oper. Res.* **18**, 1–11 (1993)

[PaCo80] Pape, U., Conradt, D.: Maximales Matching in Graphen. In: Späth, H. (ed) *Ausgewählte Operations Research Software in FORTRAN*, pp. 103–114. Oldenbourg, München (1980)

[Pel36] Peltesohn, R.: *Das Turnierproblem für Spiele zu je dreien.* Dissertation, Universität Berlin (1936)

[Pet91] Petersen, J.: Die Theorie der regulären Graphen. *Acta Math.* **15**, 193–220 (1891)

[Pet98] Petersen, J.: Sur le théorème de Tait. *L'Intermed. de Mathémat.* **5**, 225–227 (1898)

[PeLo88] Peterson, P.A., Loui, M.C.: The general maximum matching algorithm of Micali and Vazirani. *Algorithmica* **3**, 511–533 (1988)

[Plu94] Plummer, M.D.: Extending matchings in graphs: a survey. *Discr. Math.* **127**, 277–292 (1994)

[Plu96] Plummer, M.D.: Extending matchings in graphs: an update. *Congr. Numer.* **116**, 3–32 (1996)

[Pos69] Posner, E.C.: Combinatorial structures in planetary reconnaissance. In: *Error Correcting Codes* (Ed. H. B. Mann), pp. 15–46. Wiley, New York (1969).

[Pri57] Prim, R.C.: Shortest connection networks and some generalizations. *Bell Systems Techn. J.* **36**, 1389–1401 (1957)

[Pri96] Prisner, E.: Line graphs and generalizations – a survey. *Congr. Numer.* **116**, 193–229 (1996)

[PrSt02] Prömel, H.-J., Steger, A.: *The Steiner Tree Problem.* Vieweg, Braunschweig (2002)

[Pro86] Provan, J.S.: The complexity of reliability computations in planar and acyclic graphs. *SIAM J. Comp.* **15**, 694–702 (1986)

[Pru18] Prüfer, H.: Neuer Beweis eines Satzes über Permutationen. *Arch. Math. und Physik (3)* **27**, 142–144 (1918)

[Pul83] Pulleyblank, W.R.: Polyhedral combinatorics. In: Bachem, A., Grötschel, M., Korte, B. (eds) *Mathematical Programming: The State of the Art*, pp. 312–345. Springer, Berlin (1983)

[PyPe70] Pym, J.S., Perfect, H.: Submodular functions and independence structures. *J. Math. Anal. Appl.* **30**, 1–31 (1970)

[Qi88] Qi, L.: Directed submodularity, ditroids and directed submodular flows. *Math. Progr.* **42**, 579–599 (1988)

[Rad42] Rado, R.: A theorem on independence relations. *Quart. J. Math.* **13**, 83–89 (1942)

[Rad57] Rado, R.: Note on independence functions. *Proc. London Math. Soc.* **7**, 300–320 (1957)

[RaGo91] Radzik, T., Goldberg, A.V.: Tight bounds on the number of minimum mean cycle cancellations and related results. In: *Proc.* 2^{nd} *ACM – SIAM Symp. on Discrete Algorithms*, pp. 110–119 (1991)

[Ral81] Ralston, A.: A new memoryless algorithm for de Bruijn sequences. *J. Algor.* **2**, 50–62 (1981)

[Ram68] Ramachandra Rao, A.: An extremal problem in graph theory. *Israel J. Math.* **6**, 261–266 (1968)

[RaWi71] Ray-Chaudhuri, D.K., Wilson, R.M.: Solution of Kirkman's school girl problem. In: *Proc. Symp. Pure Appl. Math.* **19**, pp. 187–203. Amer. Math. Soc., Providence, R.I. (1971)

[Rea62] Read, R.C.: Euler graphs on labeled nodes. *Canad. J. Math.* **14**, 482–486 (1962).

[Rec89] Recski, A.: *Matroid Theory and its Applications.* Springer, Berlin (1989)

[Red34] Redéi, L.: Ein kombinatorischer Satz. *Acta Litt. Szeged* **7**, 39–43 (1934)

[Ree87] Rees, R.: Uniformly resolvable pairwise balanced designs with block sizes two and three. *J. Comb. Th. (A)* **45**, 207–225 (1987)

[Rei94] Reinelt, G.: *The Traveling Salesman: Computational Solutions for TSP Applications.* Springer, Berlin (1994)

[ReTa81] Reingold, E.M., Tarjan, R.E.: On a greedy heuristic for complete matching. *SIAM J. Computing* **10**, 676–681 (1981)

[Ren59] Rényi, A.: Some remarks on the theory of trees. *Publ. Math. Inst. Hungar. Acad. Sc.* **4**, 73–85 (1959)

[Rie91] Rieder, J.: The lattices of matroid bases and exact matroid bases. *Archiv. Math.* **56**, 616–623 (1991)

[Riz00] Rizzi, R.: A short proof of König's matching theorem. *J. Graph Th.* **33**, 138–139 (2000)

[Rob39] Robbins, H.: A theorem on graphs with an application to a problem of traffic control. *Amer. Math. Monthly* **46**, 281–283 (1939)

[RoXu88] Roberts, F.S., Xu, Y.: On the optimal strongly connected orientations of city street graphs I: Large grids. *SIAM J. Discr. Math.* **1**, 199–222 (1988)

[RoSST97] Robertson, N., Sanders, D.P., Seymour, P., Thomas, R.: The four-colour theorem. *J. Comb. Th. (B)* **70**, 2–44 (1997)

[RoST93] Robertson, N., Seymour, P., Thomas, R.: Hadwiger's conjecture for K_6-free graphs. *Combinatorica* **13**, 279–361 (1993)

[RoSL77] Rosenkrantz, D.J., Stearns, E.A., Lewis, P.M.: An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comp.* **6**, 563–581 (1977)

[Rob03a] Robinson, S.: Are medical students meeting their (best possible) match? *SIAM News* **36 (3)**, 8–9 (2003)

[Rob03b] Robinson, S.: How much can matching theory improve the lot of medical residents? *SIAM News* **36 (6)**, 4–5 (2003)

[Ros77] Rosenthal, A.: Computing the reliability of complex networks. *SIAM J. Appl. Math.* **32**, 384–393 (1977)

[Rue86] Rueppel, R.: *Analysis and Design of Stream Ciphers.* Springer, New York (1986)

[Rys57] Ryser, H.J.: Combinatorial properties of matrices of zeros and ones. *Canad. J. Math.* **9**, 371–377 (1957)

[SaGo76] Sahni, S., Gonzales, T.: P-complete approximation problems. *J. Ass. Comp. Mach.* **23**, 555–565 (1976)

[Schn78] Schnorr, C.P.: An algorithm for transitive closure with linear expected time. *SIAM J. Comp.* **7**, 127–133 (1978)

[Schn79] Schnorr, C.P.: Bottlenecks and edge connectivity in unsymmetrical networks. *SIAM J. Comp.* **8**, 265–274 (1979)

[Schr80] Schreuder, J.A.M.: Constructing timetables for sport competitions. *Math. Progr. Studies* **13**, 58–67 (1980)

[Schr92] Schreuder, J.A.M.: Combinatorial aspects of construction of competition Dutch professional football leagues. *Discr. Appl. Math.* **35**, 301–312 (1992)

[Schr83a] Schrijver, A.: Min-max results in combinatorial optimization. In: Bachem, A., Grötschel, M., Korte, B. (eds) *Mathematical Programming: The State of the Art*, pp. 439–500. Springer, Berlin (1983)

[Schr83b] Schrijver, A.: Short proofs on the matching polyhedron. *J. Comb. Th. (B)* **34**, 104–108 (1983)

[Schr84] Schrijver, A.: Total dual integrality from directed graphs, crossing families, and sub- and supermodular functions. In: Pulleyblank, W.R. (ed) *Progress in Combinatorial Optimization*, pp. 315–361. Academic Press Canada (1984)

[Schr86] Schrijver, A.: *Theory of Integer and Linear Programming.* Wiley, New York (1986)

[Schr03] Schrijver, A.: *Combinatorial Optimization. Polyhedra and Efficiency (in 3 volumes).* Springer, Berlin (2003)

[Schw91] Schwenk, A.J.: Which rectangular chessboards have a knight's tour? *Math. Magazine* **64**, 325–332 (1991)

[ScWi78] Schwenk, A.J., Wilson, R.: On the eigenvalues of a graph. In: Beineke, L., Wilson, R. (eds) *Selected Topics in Graph Theory*, pp. 307–336. Academic Press, London (1978)

[Sey79] Seymour, P.: Sums of circuits. In: Bondy, J.A., Murty, U.S.R. (eds) *Graph Theory and Related Topics*, pp. 341–355. Academic Press, New York (1979)

[Sha48] Shannon, C.E.: A mathematical theory of communication. *Bell Syst. Tech. J.* **27**, 379–423 (Part I) and 623–656 (Part II) (1948)

[Sha49a] Shannon, C.E.: A theorem on colouring lines of a network. *J. Math. Phys.* **28**, 148–151 (1949)

[Sha49b] Shannon, C.E.: Communication theory of secrecy systems. *Bell Syst. Tech. J.* **28**, 656–715 (1949)

[Sha79] Shapiro, J.F.: A survey of Langrangian techniques for discrete optimization. *Ann. Discr. Math.* **5**, 113–138 (1979)

[Shi75] Shimbel, A.: Structure in communication nets. In: *Proc. Symp. Information Networks*, pp. 199–203. Polytechnic Institute of Brooklyn, New York (1955)

[ShWi90] Shmoys, D.B., Williamson, D.P.: Analyzing the Held-Karp-TSP bound: A monotonicity property with application. *Inform. Proc. Letters* **35**, 281–285 (1990)

[Sho85] Shor, N.Z.: *Minimization Methods for Non-Differentiable Functions.* Springer, Berlin (1985)

[SiHo91] Sierksma, G., Hoogeveen, H.: Seven criteria for integer sequences being graphic. *J. Graph Th.* **15**, 223–231 (1991)

[SiTu89] Siklóssy, L., Tulp, E. (1989): Trains, an active time-table searcher. *ECAI '89*, 170–175 (1991)

[Sim88] Simões-Pereira, J.M.S.: An optimality criterion for graph embeddings of metrics. *SIAM J. Discr. Math.* **1**, 223–229 (1988)

[Sle80] Sleator, D.D.: *An $O(mn \log n)$ algorithm for maximum network flow.* Ph.D. thesis, Stanford University (1980)

[SlTa83] Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. *J. Comput. System Sci.* **26**, 362–391 (1983)

[Spe28] Sperner, E.: Ein Satz über Untermengen einer endlichen Menge. *Math. Z.* **27**, 544–548 (1928)

[Spi85] Spinrad, J.: On comparability and permutation graphs. *SIAM J. Comp.* **14**, 658–670 (1985)

[Sta86] Stanley, R.P.: *Enumerative Combinatorics Vol. 1.* Wadsworth & Brooks/Cole, Monterey (1986)

[Sta99] Stanley, R.P.: *Enumerative Combinatorics Vol. 2.* Cambridge University Press, Cambridge (1999)

[StLe80] Stern, G., Lenz, H.: Steiner triple systems with given subspaces: Another proof of the Doyen-Wilson theorem. *Bolletiono U.M.I. (5)* **17**, 109–114 (1980)

[Sto85] Stong, R.A.: On 1-factorizability of Cayley graphs. *J. Comb. Th. (B)* **39**, 298–307 (1985)

[Str88] Strang, G.: *Linear Algebra and its Applications (Third Edition).* Harcourt Brace Jovanovich, San Diego (1993)

[Sum79] Sumner, D.P.: Randomly matchable graphs. *J. Graph Th.* **3**, 183–186 (1979)

[Suz82] Suzuki, M.(1982): *Group Theory I.* Springer, Berlin–Heidelberg–New York (1979)

[SyDK83] Syslo, M.M., Deo, N., Kowalik, J.S.: *Discrete Optimization Algorithms.* Prentice Hall, Englewood Cliffs, N.J. (1983)

[Ta92] Taha, H.A.: *Operations Research (Fifth Edition).* Macmillan Publishing Co., New York (1992)

[Tak90a] Takács, L.: On Cayley's formula for counting forests. *J.Comb.Th. (A)* **53**, 321–323 (1990)

[Tak90b] Takács, L.: On the number of distinct forests. *SIAM J. Discr. Math.* **3**, 574–581 (1990)

[Tak92] Takaoka, T.: A new upper bound on the complexity of the all pairs shortest path problem. *Inf. Process. Lett.* **43**, 195–199 (1992)

[Tar85] Tardos, E.: A strongly polynomial minimum cost circulation algorithm. *Combinatorica* **5**, 247–255 (1985)

[Tar86] Tardos, E.: A strongly polynomial algorithm to solve combinatorial linear programs. *Oper. Res.* **34**, 250–256 (1986)

[Tar72] Tarjan, R.E.: Depth first search and linear graph algorithms. *SIAM J. Comp.* **1**, 146–160 (1972)

[Tar77] Tarjan, R.E.: Finding optimum branchings. *Networks* **7**, 25–35 (1977)

[Tar83] Tarjan, R.E.: *Data Structures and Network Algorithms.* Soc. Ind. Appl. Math., Philadelphia (1983)

[Tar84] Tarjan, R.E.: A simple version of Karzanov's blocking flow algorithm. *Oper. Res. Letters* **2**, 265–268 (1984)

[Tar97] Tarjan, R.E.: Dynamic trees as search trees via Euler tours applied to the network simplex algorithm. *Math. Progr.* **78** (1997), 169–177 (1997)

[Tar95] Tarry, G.: Le problème des labyrinthes. *Nouv. Ann. de Math.* **14**, 187 (1895)

[Tas97] Tassiulas, L.: Worst case length of nearest neighbor tours for the euclidean traveling salesman problem. *SIAM J. Discr. Math.* **10**, 171–179 (1997)

[Ter96] Terlaky, T.: *Interior Point Methods of Mathematical Programming.* Kluwer, Dordrecht (1996)

[Tho98] Thomas, R.: An update on the four-color theorem. *Notices Amer. Math. Soc.* **45**, 848–859 (1998)

[Tho81] Thomassen, C.: Kuratowski's theorem. *J. Graph Th.* **5**, 225–241 (1981)

[Tho94] Thomassen, C.: Every planar graph is 5-choosable. *J. Comb. Th. (B)* **62**, 180–181 (1994)

[Tof96] Toft, B.: A survey of Hadwiger's conjecture. *Congr. Numer.* **115**, 249–283 (1996)

[TrHw90] Trietsch, D., Hwang, F.: An improved algorithm for Steiner trees. *SIAM J. Appl. Math.* **50**, 244–264 (1990)

[Tur36] Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc. (2)* **42**, 230–265 (1936)

[Tur88] Turner, J.S.: Almost all k-colorable graphs are easy to color. *J. Algor.* **9**, 63–82 (1988)

[Tut47] Tutte, W.T.: The factorization of linear graphs. *J. London Math. Soc.* **22**, 107–111 (1947)

[Tut48] Tutte, W.T.: The dissection of equilateral triangles into equilateral triangles. *Proc. Cambridge Phil. Soc.* **44**, 203–217 (1948)

[Tut52] Tutte, W.T.: The factors of graphs. *Canad. J. Math.* **4**, 314–328 (1952)

[Tut54] Tutte, W.T.: A short proof of the factor theorem for finite graphs. *Canad. J. Math.* **6**, 347–352 (1952)

[Tut67] Tutte, W.T.: Antisymmetrical digraphs. *Canad. J. Math* **19**, 1101–1117 (1967).

[Tut71] Tutte, W.T.: *Introduction to the Theory of Matroids.* Elsevier, New York (1971)

[Tut84] Tutte, W.T.: *Graph Theory.* Cambridge University Press, Cambridge (1984)

[Vai89] Vaidya, P.M.: Geometry helps in matching. *SIAM J. Comput.* **19**, 1201–1225 (1989)

[Val79a] Valiant, L.G.: The complexity of computing the permanent. *Theor. Comp. Sc.* **8**, 189–201 (1979)

[Val79b] Valiant, L.G. : The complexity of enumeration and reliability problems. *SIAM J. Comp.* **8**, 410–421 (1979)

[vdW26] van der Waerden, B.L.: Aufgabe 45. *Jahresber. DMV*, 117 (1926)

[vdW27] van der Waerden, B.L.: Ein Satz über Klasseneinteilungen von endlichen Mengen. *Abh. Math. Sem. Hamburg* **5**, 185–188 (1927)

[vdW37] van der Waerden, B.L.: *Moderne Algebra (Zweite Auflage).* Springer, Berlin (1937)

[vdW49] van der Waerden, B.L.: *Modern algebra, Vol. I* (translated from the 2nd revised German edition by Fred Blum, with revisions and additions by the author). Frederick Ungar Publishing Co., New York (1949)

[vLi74] van Lint, J.H.: *Combinatorial Theory Seminar Eindhoven University of Technology.* Springer, Berlin (1974)

[vLi99] van Lint, J.H.: *Introduction to Coding Theory (Third Edition).* Springer, Berlin– Heidelberg–New York (1999)

[vLiWi01] van Lint, J.H., Wilson, R.M.: *A Course in Combinatorics (Second Edition).* Cambridge University Press, Cambridge (2001)

[VaOo89] Vanstone, S.A., van Oorschot, P.C.: *An Introduction to Error Correcting Codes with Applications.* Kluwer, Boston (1989).

[Vaz94] Vazirani, V.V.: A theory of alternating paths and blossoms for proving correctness of the $O(V^{1/2}E)$ general graph matching algorithm. *Combinatorica* **14**, 71–109 (1994)

[Viz64] Vizing, V.G.: An estimate of the chromatic class of a p-graph (in Russian). *Diskret. Analiz.* **3**, 25–30 (1964)

[Voi93] Voigt, M.: List colourings of planar graphs. *Discr. Math.* **120**, 215–219 (1993)

[VoJo82] Volgenant, T., Jonker, R.: A branch and bound algorithm for the symmetric travelling salesman problem based on the 1-tree relaxation. *Europ. J. Oper. Res.* **9**, 83–89 (1982)

[Vol04] Volkmann, L.: The Petersen graph is not 1-factorable: postscript to 'The Petersen graph is not 3-edge-colorable – a new proof' *Discr. Math.* **287**, 193–194 (2004)

[Vos92] Voß, S.: Steiner's problem in graphs: heuristic methods. *Discr. Appl. Math.* **40**, 45–72 (1992)

[Wag36] Wagner, K.: Bemerkungen zum Vierfarbenproblem. *Jahresber. DMV* **46**, 26–32 (1936)

[Wag37] Wagner, K.: Über eine Eigenschaft der ebenen Komplexe. *Math. Ann.* **114**, 170–190 (1937)

[Wag60] Wagner, K.: Bemerkungen zu Hadwigers Vermutung. *Math. Ann.* **141**, 433–451 (1960)

[Wal92] Wallis, W.D.: One-factorizations of the complete graph. In: Dinitz, J.H., Stinson, D.R. (eds) *Contemporary Design Theory: A Collection of Surveys*, pp. 593–639. Wiley, New York (1992)

[Wal97] Wallis, W.D.: *One-Factorizations.* Kluwer Academic Publishers, Dordrecht (1997)

[War62] Warshall, S.: A theorem on Boolean matrices. *J. Ass. Comp. Mach.* **9**, 11–12 (1962)

[Wei74] Weintraub, A.: A primal algorithm to solve network flow problems with convex costs. *Management Sc.* **21**, 87–97 (1974)

[Wel68] Welsh, D.J.A.: Kruskal's theorem for matroids. *Proc. Cambridge Phil. Soc.* **64**, 3–4 (1968)

[Wel76] Welsh, D.J.A.: *Matroid Theory.* Academic Press, New York (1976)

[Whi86] White, N. (ed): *Theory of Matroids.* Cambridge University Press, Cambridge (1986)

[Whi87] White, N. (ed): *Combinatorial Geometries.* Cambridge University Press, Cambridge (1987)

[Whi92] White, N. (ed): *Matroid Applications.* Cambridge University Press, Cambridge (1992)

[Whi32a] Whitney, H.: Congruent graphs and the connectivity of graphs. *Amer. J. Math.* **54**, 150–168 (1932)

[Whi32b] Whitney, H.: Non-separable and planar graphs. *Trans. Amer. Math. Soc.* **54**, 339–362 (1932)

[Whi33] Whitney, H.: Planar graphs. *Fund. Math.* **21**, 73–84 (1933)

[Whi35] Whitney, H.: On the abstract properties of linear dependence. *Amer. J. Math.* **57**, 509–533 (1935)

[Wil72] Wilson, L.B.: An analysis of the stable marriage assignment problem. *BIT* **12**, 569–575 (1972)

[Wil86] Wilson, R.J.: An Eulerian trail through Königsberg. *J. Graph Th.* **10**, 265–275 (1986)

[Wil89] Wilson, R.J.: A brief history of Hamiltonian graphs. *Ann. Discr. Math.* **41**, 487–496 (1989)

[Wil2002] Wilson, R.J.: *Four Colors Suffice: How the Map Problem was Solved.* Princeton University Press, Princeton (2002)

[Win88] Winkler, P.: The complexity of metric realization. *SIAM J. Discr. Math.* **1**, 552–559 (1988)

[Wir76] Wirth, N.: *Algorithms + Data Structures = Programs.* Prentice Hall, Englewood Cliffs, N.J. (1976)

[Wol80] Wolsey, L.A.: Heuristic analysis, linear programming and branch and bound. *Math. Progr. Stud.* **13**, 121–134 (1980

[Woo01] Woodall, D.R.: List colourings of graphs. In: Hirschfeld, J.W.P. (ed) *Combinatorial Surveys*, pp. 269–301. Cambridge University Press, Cambridge (2001))

[Yan78] Yannakakis, M.: Node- and edge-deletion NP-complete problems. In: *Proc. 10th ACM Symp. on Theory of Computing*, pp. 253–264. Ass. Comp. Mach., New York (1978)

[YaGa80] Yannakakis, M., Gavril, F.: Edge dominating sets in graphs. *SIAM J. Appl. Math.* **38**, 364–372 (1980)

[Yao75] Yao, A.C.: An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees. *Inform. Proc. Letters* **4**, 21–23 (1975)

[Yap86] Yap, H.P.: *Some Topics in Graph Theory.* Cambridge University Press, Cambridge (1986)

[YoTO91] Young, N.E., Tarjan, R.E., Orlin, J.B.: Faster parametric shortest path and minimum-balance algorithms. *Networks* **21**, 205–221 (1991)

[Yu98] Yu, G. (ed.): *Operations Research in the Airline Industry.* Springer, New York (1998)

[YuZw97] Yuster, R., Zwick, U.: Finding even cycles even faster. *SIAM J. Discr. Math.* **10**, 209–222 (1997)

[Zad72] Zadeh, N.: Theoretical efficiency of the Edmonds-Karp algorithm for computing maximal flows. *J. Ass. Comp. Mach.* **19**, 248–264 (1972)

[Zad73a] Zadeh, N.: A bad network problem for the simplex method and other minimum cost flow algorithms. *Math. Progr.* **5**, 255–266 (1973)

[Zad73b] Zadeh, N.: More pathological examples for network flow problems. *Math. Progr.* **5**, 217–224 (1973)

[Zim81] Zimmermann, U.: *Linear and Combinatorial Optimization in Ordered Algebraic Structures.* North Holland, Amsterdam (1981)

[Zuc96] Zuckerman, D.: On unapproximable versions of NP-complete problems. *SIAM J. Computing* **25**, 1293–1304 (1996)

Index

We look before and after;
We pine for what is not...
Percy Bisshe Shelley

accessibility axiom, 148
accessible, 26
accessible set system, 148
active vertex, 192, 315
acyclic digraph, 46, 72, 76, 228, 253
acyclic graph, 7
adjacency list, 38
adjacency matrix, 38, 586
adjacent, 2, 497
admissible cell, 223
admissible edge, 320
admissible flow, 153, 344
admissible graph, 320
admissible operations, 316
admissible PUSH, 192, 316
admissible RELABEL, 192, 316
admissible vector, 431
algebraic assignment problem, 429
algorithm, 33, 34, 41–46
  ε-approximative, 471–477
  dual greedy, 145
  efficient, 45
  farthest insertion, 478
  FIFO preflow push, 198
  good, 45
  greedy, 128
  highest label preflow push, 200
  Hungarian, 421–430
  insertion, 478
  labelling, 157, 294
  local search, 480
  minimum mean cycle-canceling, 323
  MKM, 179–182
  nearest insertion, 479
  polynomial, 45
  primal-dual, 434
  strongly polynomial, 298
  tree, 472
  weakly polynomial, 298
algorithm of
  Bellman and Ford, 80
  Boruvka, 111
  Busacker and Gowen, 299–302, 383–386
  Christofides, 474
  Dijkstra, 76–80
  Dinic, 176, 186
  Edmonds, 409
  Edmonds and Karp, 159–169
  Floyd and Warshall, 84–89
  Ford and Fulkerson, 157, 294
  Goldberg and Tarjan, 189–203, 311–322
  Gusfield, 366
  Hierholzer, 35, 39–40
  Klein, 295–298, 322–327
  Kruskal, 109
  Malhotra, Kumar and Mahaswari, 179–182
  Minty, 294
  Moore, *see* BFS
  Prim, 107
  Tarjan, 250
alphabet, 40, 329
alternating forest, 424
alternating path, 390

alternating tree, 395
ancestor, 244
and, 42
antichain, 228
antiparallel, 25
apex, 355
approximation scheme, 476
ε-approximative algorithm, 471–477
arbitrarily traceable, 36
arborescence, 27
  spanning, 69, 121–125, 244
arc, 25
  entering, 354
  leaving, 354
articulation point, 245
assignment (of values to variables), 42
assignment problem, 227, 281
assignment relaxation, 461, 497
associated digraph, 25
asymmetric travelling salesman problem, 490
asymmetric TSP, 459
ATSP, 459, 476
AUGMENT, 175, 360, 412, 424
augmenting path, 155, 351, 390
augmenting path theorem, 155, 391
automorphism, 24
automorphism group, 24, 271
  regular, 271
auxiliary netork, 169
auxiliary network
  layered, 172
AUXNET, 174

back edge, 244, 253
backward adjacency list, 38
backward edge
  in a directed path, 26
  in a flow network, 155
balanced flow, 387
balanced network, 387
Baranyai's theorem, 231
base, 401
basis, 131, 148
basis completion theorem, 132
BELLFORD, 80
Bellman's equations, 70, 90
BFS, 63–68
biconnected component, *see* block
BIPART, 67
bipartite graph, 65, 103, 188, 213, 226, 236, 261, 399, 416, 420–430
  complete, 2
  regular, 216
  symmetric, 416
BIPMATCH, 399
block, 246–252
block-cutpoint graph, 247
BLOCK01FLOW, 186
BLOCKCUT, 250
BLOCKFLOW, 176
blocking flow, 171–183, 186
BLOCKMKM, 179
blossom, 397–408
BLOSSOM, 411
bond space, 292
Boolean variable, 50
border, 275
BORUVKA, 111
bottleneck assignment problem, 226, 429
bottleneck problem, 114, 150
branch and bound, 489–497
branching, 27
breadth first search, *see* BFS
break, 29
bridge, 22, 27, 290
Brooks' theorem, 263

cancelling a cycle, 296
canonical, 31
capacity, 345
  in a digraph, 279
  in a flow network, 153
  of a cut, 154, 287
  of a path, 114
  residual, 191, 305
capacity constraints, 279
capacity function, 363
capacity increase, 383–386
capacity restrictions, 344
cardinality matching problem, 419
caterer problem, 282
Cayley graph, 271
cell (of a matrix), 223
  admissible, 223
center, 88
certificate, 51

chain, 228
k-change neighborhood, 480
Chinese postman problem, 438–442, 501
choosability, 276
chord, 266
chordal, 266
Christofides' algorithm, 474
chromatic index, 268, 501
chromatic number, 261, 339, 501
circuit, 133
circuit axioms, 133
circulation, 279, 304
  ε-optimal, 306
  ε-tight, 308
  elementary, 290, 293
  feasible, 279, 304
  legal, 279
  minimum cost, 280
  optimal, 280, 304
circulation theorem, 288
clause, 50
clique, 55, 502
clique number, 266
clique partition number, 266
clique problem, 55
closed set, 132
closed trail, 5
closed walk, 5
closure, 16
  hereditary, 150
  transitive, 85
closure congruence axiom, 150
Co-NP, 51
cobasis, 135
cocircuit, 135, 293
cocycle, 105
  directed, 293
cocycle space, 292
code, 329
  t-error correcting, 331
  augmented Petersen, 337, 455
  binary, 329
  cyclic, 455
  decoding algorithms, 452, 454
  even graphical, 332
  extended binary Hamming, 339
  linear, 331
  odd pattern, 338
  parameters, 331
  parity check extension, 339
  Petersen, 333
  purely graphical, 340
codeword, 329
color, 261
COLOR, 262
coloring, 261
  edge, 268
common system of representatives, 222
common transversal, 222
COMP, 520
comparability graph, 265
comparable, 265
complementary graph, 5
complementary slackness conditions, 306
complete bipartite graph, 2
complete digraph, 25
complete graph, 2
complete matching, 215
complete orientation, 25
complete time-sharing, 378
complexity, 44
component
  biconnected, *see* block
  connected, 6, 65, 243, 272
  odd, 388
  strong, 253–257
condensation, 257
conjecture
  four color, 278
  Steiner ratio, 116
  strong perfect graph, 268
conjecture of
  Berge, 267
  Hadwiger, 265
  Hajós, 265
  Lovász, 275
  Sylvester, 234
  van der Waerden, 225
conjunctive normal form, 50
connected component, 6, 65, 243, 272
  strong, 253–257
connected digraph, 26
  strongly, 26, 253–257
connected graph, 6, 65
  2-connected, 245–252
  m-fold edge, 258
  k-connected, 212, 239–242

connected vertices, 6
connectivity, 212, 239–242
connector problem, 97
constraints, 430
CONTRACT, 411
contractible, 24
contraction, 23, 400
  elementary, 23
convex function, 302
convex hull, 225
Cook's theorem, 52
cost
  for capacity increase, 383–386
  of a circulation, 279
  of a flow, 281
  of a matching, 420
  of a pseudoflow, 304
cost curve, 302
cost function, 344
  for capacity increase, 383–386
  for circulations, 279
  reduced, 305
cotree, 135
cover (a matching covers a set), 221, 416, 417
CPP, *see* Chinese postman problem, 439, 501
critical edge, 160
critical path, 73
critical path method, 72
critical subfamily, 219
critical task, 73
cross edge, 253
crossing cuts, 369
current edge, 197, 320
current vertex, 320
cut, 345
  in a digraph, 287
  in a flow network, 154
  in a graph, 105
  minimal, 154
cut point, 245
cut tree, 379–383
cuts
  crossing, 369
  non-crossing, 369
CUTTREE, 380
cycle, 5
  cancelling, 296
  directed, 26
  Hamiltonian, 15, 52
    directed, 52
  of minimum cycle mean, 308–311, 323–327
  of negative length, 89–90, 297, 305, 307
  pivot, 354
  shortest, 68
cycle mean
  minimum, 308
cycle space, 291, 332
cyclic graph, 273
cyclomatic number, 291

dag, *see* acyclic digraph
de Bruijn sequence, 40, 124
DECAUGGC, 454
DECEVGC, 452
decision problem, 49
decision tree, 489
decomposition theorem, 224
deficiency, 219
deficiency version
  of the 1-factor theorem, 389
  of the marriage theorem, 219
degree, 4
degree matrix, 102
degree sequence, 12
DELETEMIN, 79
demand, 235
demand function, 344
demand restrictions, 344
dense, 45
dependent set, 131
depth first search, *see* DFS
depth index, 358
descendant, 244
determinism, 34
DFS, 242–245, 256
DFSM, 255
DHC, *see* directed Hamiltonian cycle problem
DHP, *see* directed Hamiltonian path problem
diagonal, 224–227
  non-zero, 224
  positive, 224
diameter, 88, 502

digraph, 25–28, 46–49, 99–102, 121–125, 153, 227–231, 252–257, 279
  acyclic, 46–49, 72, 76, 228, 253
  associated, 25
  complete, 25
  condensation, 257
  connected, 26
  layered, 172
  pseudosymmetric, 26
  strongly connected, 26, 253–257
  symmetric, 190
  transitive, 85
  transitive reduction, 87
DIJKSTRA, 76
DIJKSTRAPQ, 79
Dilworth number, 229
Dilworth's theorem, 228
dioid, 91
directed cocycle, 293
directed cycle, 26
directed Euler tour, 26
directed graph, *see* digraph
directed Hamiltonian cycle, 52
directed Hamiltonian cycle problem, 52
directed Hamiltonian path, 142, 143
directed Hamiltonian path problem, 143
directed multigraph, 25
directed path, 26
directed trail, 26, 27
directed tree, *see* arborescence
Discrete metric realization, 502
dissection, 228
distance, 7, 60–65
  Hamming, 331
  to a partial tour, 478
distance matrix, 62, 84
dodecahedral graph, 15
dominant requirement tree, 373
dominating network, 377
doubly stochastic matrix, 224
dual
  geometric, 136
dual greedy algorithm, 145
dual linear program, 432
dual linear programming problem, 432
dual matroid, 135–137, 144
DUALGREEDY, 145
duality theorem, 433
dynamic flow, 329
easy problem, 45, 49
edge, 2, 25
  ε-fixed, 313
  admissible, 320
  antiparallel, 25
  back, 244, 253
  backward, 26
  connected, 258
  critical, 160
  cross, 253
  current, 197, 320
  forward, 26, 253
  free, 347
  most vital, 159
  originating, 202
  parallel, 13
  residual, 191, 305
  saturated, 154
  tree, 244, 252
  void, 154
m-fold edge connected, 258
edge chromatic number, 268
edge coloring, 268
edge connectivity, 258
edge disjoint paths, 209, 503
edge list, 36
edge separator, 209
edge set, 2
effectiveness, 34
efficiency, 34
efficient, 45
elementary circulation, 290, 293
elementary contraction, 23
elementary flow, 159
end vertex, 2, 5, 25
EPM, *see* exact perfect matching problem
equality subgraph, 421
equimatchable, 387
equivalent flow tree, 365
error, 331
error pattern, 331
Euclidean Steiner problem, 115
Euclidean TSP, 477, 482
EULER, 40, 43
Euler tour, 13, 39–41
  directed, 26
Euler's formula, 21

Eulerian cycle, *see* Euler tour
Eulerian graph, 13, 39–41
Eulerian multigraph, 13
  spanning, 472
Eulerian trail, 13
evaluation problem, 50
even path, 447
even vertex, 396
exact neighborhood, 484
exact perfect matching problem, 450
excentricity, 87
exchange axiom, 131, 132
  strong, 149
exposed vertex, 389
extensibility axiom, 150

face, 21
factor, 4
  1-, 4
  2-, 216
  f-, 417
  k-, 4
  $\triangle$-, 216
  triangle, 216
1-factor theorem, 387
  deficiency version, 389
factorization, 4
  1-, 4
  2-, 216
  k-, 4
  oriented, 28–31
FAREFINE, 321
FARIN, 478
farthest insertion algorithm, 478
feasibility condition, 153
feasible circulation, 279, 304
feasible flow, 235
feasible network, 365
feasible node-weighting, 421
feasible set, 148
Fermat point, 115
FIFO preflow push algorithm, 198
FIFOFLOW, 198
finiteness of description, 34
first active method, 321
first improvement, 480
five color theorem, 275, 278
ε-fixed, 313
float, 73
flow, 153, 344
  0-1-, 185
  blocking, 171–183, 186
  dynamic, 329
  elementary, 159
  feasible, 235
  maximal, 154
  minimal feasible, 289
  multicommodity, 329, 378
  optimal, 281, 299, 328
flow conservation condition, 154
flow excess, 191
flow function, 363
flow network, 153
  layered, 172–183, 186
flow potential, 179
flow with gain or loss, 329
FLOWTREE, 367
FLOYD, 84
for ... **do**, 42
FORDFULK, 157
forest, 8, 100
  minimal spanning, 104
  alternating, 424
forward edge
  in a DFS, 253
  in a directed path, 26
  in a flow network, 155, 211
four color conjecture, 278
four color theorem, 278
free matroid, 129

Gale-Ryser theorem, 236
generalized dihedral group, 274
generating set, 132, 135
geometric dual, 136
geometric graph, 21
geometric Steiner tree problem, 115
girth, 22
GOBLIN, 353
GOLDBERG, 192
good algorithm, 45
gozinto graph, 94
graph, 2
  2-connected, 245–252
  m-fold edge connected, 258
  acyclic, 7
  admissible, 320
  arbitrarily traceable, 36

bipartite, 65, 103, 188, 213, 226, 236, 261, 399, 416, 420–430
symmetric, 416
block-cutpoint, 247
Cayley, 271
chordal, 266
comparability, 265
complementary, 5
complete K_n, 2
complete bipartite $K_{m,n}$, 2
connected, 6, 65
contracted, 400
contractible, 24
cyclic, 273
dense, 45
directed, *see* digraph
dodecahedral, 15
edge connected, 258
equality, 421
equimatchable, 387
Eulerian, 13, 39–41
geometric, 21
gozinto, 94
Hamiltonian, 15
homeomorphic, 23
interval, 266
isomorphic, 21
k-connected, 212, 239–242
line, 14
mixed, 27
orientable, 27
perfect, 267
Petersen, 24, 216, 265, 271, 333, 548, 579
planar, 21–25
plane, 21
randomly matchable, 394
regular, 4
regular bipartite, 216
residual, 191, 305
separable, 246
sparse, 45
strongly regular, 5, 39
triangular T_n, 5
triangulated, 266
underlying, 25
unicyclic, 98
uniform, 373
graph partitioning, 503
graphic matroid, 129
graphical code, 336
greedoid, 149
GREEDY, 128, 148
greedy algorithm, 128
dual, 145

Hadwiger's conjecture, 265
Hajós' conjecture, 265
Hamiltonian cycle, 15, 52
directed, 52
Hamiltonian cycle problem, 49, 503
Hamiltonian graph, 15
Hamiltonian path, 52, 143
directed, 142, 143
Hamiltonian path problem, 52, 503
Hamming distance, 331
hard problem, 45
harem theorem, 223
Hasse diagram, 87
HC, *see* Hamiltonian cycle problem
head, 25
head-partition matroid, 129
heap, 79
hereditary closure, 150
heuristics, 477–479
highest label preflow push algorithm, 200
Hitchcock problem, 328
HLFLOW, 200
home-away pattern, 29
homeomorphic, 23
HP, *see* Hamiltonian path problem
HUNGARIAN, 423
Hungarian algorithm, 421–430
hyperplane, 132

Icosian game, 15
if ... **then** ... **else**, 42
ILP, *see* integer linear programming problem
incidence list, 37
incidence map, 13
incidence matrix, 99
incident, 2, 25
increasing the capacities, 383–386
indegree, 26
indegree matrix, 121
independence number, 227

independence system, 128
independent set
  in a matroid, 128
  of vertices, 55, 227, 503
independent set of cells, 223
independent set problem, 55
induced subgraph, 3
induced subgraph problem, 504
inf-section, 114
INMATCH, 395
inner vertex, 396
insertion algorithm, 478
instance, 34
integer linear program, 431
integer linear programming problem, 431, 504
integral flow theorem, 156
intermediate node, 235, 328
intersection of matroids, 141–143
interval graph, 266
intractable problem, 45
IS, *see* independent set problem
isolated vertex, 6
isomorphic, 21
isotonic, 131
iteration, 42

k-connected, 212, 239–242
KAPPA, 241
Kirkman's school girl problem, 217
KLEIN, 296
knapsack problem, 61
knight's problem, 17
Königsberg bridge problem, 1
KRUSKAL, 109, 110
König's lemma, 225
König's theorem, 214

labelling algorithm, 157, 294
labelling, valid, 191
Lagrange relaxation, 470
Laplacian matrix, 102
lattice, 437
layered auxiliary network, 172
layered digraph, 172
layered flow network, 172–183, 186
leaf, 8, 10
league schedules, 30–32
legal circulation, 279
LEGCIRC, 286
lemma of
  König, 225
  Minty, 292
  Sperner, 230
length, 5, 59
level, 65
line graph, 14
linear program, 430
  0–1, 431
  dual, 432
  integer, 431
linear programming problem, *see* LP
linear span, 437
list
  adjacency, 38
  backward adjacency, 38
  color, 276
  edge, 36
  incidence, 37
list coloring, 276
list coloring number, 276
list of edges, 36
literal, 50
local search algorithm, 480
long trajectory, 203
longest cycle problem, 504
longest path, 61
longest path problem, 53, 504
loop, 13, 42
low point, 248
lower capacity, 279, 344
lower rank, 138
LP, 430
  dual, 432
LP relaxation, 465
LPD, *see* dual linear programming problem

m-fold edge connected, 258
MsT, 463
map, 275
map coloring, 275
marriage theorem, 218
  deficiency version, 219
MATCH, 188
matching, 129, 213
  complete, 215
  covering a set, 221, 417

maximal, 213
maximal weighted, 420
of maximal cardinality, 213
optimal, 420
perfect, 215, 387
product-optimal, 429
stable, 451
symmetric, 416
unextendable, 508
matching matroid, 417
matching polytope, 437
mate, 389
matric matroid, 131
matrix
0-1-, 225, 227, 237
adjacency, 38
degree, 102
distance, 62, 84
doubly stochastic, 224
incidence, 99
indegree, 121
Laplacian, 102
permutation, 224
quasi-inverse, 93
reduced, 490
totally unimodular, 101
matrix tree theorem, 101
matroid, 128
dual, 135–137, 144
free, 129
graphic, 129
head-partition, 129
matching, 417
matric, 131
partition, 221
representable, 131
restriction, 416
tail-partition, 129
transversal, 220
uniform, 129
vectorial, 131
matroid embedding axiom, 150
matroid intersection problem, 143, 504
max cut problem, 505
max-flow min-cut theorem, 156
max-flow problem, 280
MAX01FLOW, 554
MAXFLOW, 175
maximal flow, 154
maximal matching, 213
maximal spanning tree, 113–115
maximal weighted matching, 420
MAXLEGFLOW, 564
MAXMATCH, 410
maze, 245
MCFZIB, 353, 361
mean weight (of a cycle), 308
MEANCYCLE, 310, 569
MERGE, 110
metric space, 62
metric Steiner network problem, 116
metric travelling salesman problem, 459
mimimum distance, 331
mimimum weight, 332
min cut problem, 505
minimal counterexample, 214
minimal cut, 154
minimal network, 365
minimal potential, 179
minimal spanning forest, 104
minimal spanning tree, 104–112, 463, 472, 505
minimal Steiner tree, 116, 477
minimal vertex, 179
minimum k-connected subgraph problem, 505
minimum cost circulation, 280
minimum cost flow problem, 344
minimum cycle mean, 308–310, 323
minimum mean cycle canceling algorithm, 323
minimum spanning tree problem, 505
MINTREE, 106
MINTY, 294
Minty's painting lemma, 292
MIP, *see* matroid intersection problem
mixed graph, 27
mixed multigraph, 27
MKM-algorithm, 179–182
MMCC, 323
monotonic subsequence, 230
most vital edge, 159
MST relaxation, 462
multi-terminal network flow, 378
multicommodity flow, 329, 378
multigraph, 13
directed, 25
Eulerian, 13

mixed, 27
orientable, 27
spanning Eulerian, 472
strongly connected, 27
underlying, 25

nearest insertion algorithm, 479
NEGACYCLE, 89
neighborhood, 480
k-change, 480
exact, 484
neighbour, 2
network, 59
0-1-, 185
auxiliary, 169
dominating, 377
feasible, 365
flow, 153
layered, 172–183, 186
layered auxiliary, 172
minimal, 365
symmetric, 363
network flow, 505
network reliability problem, 114, 506
network synthesis, 363–386
node
intermediate, 235, 328
transshipment, 235
non-crossing cuts, 369
non-saturating PUSH, 196, 318
non-zero diagonal, 224
NP, 51
NP-complete problem, 46, 51
NP-hard problem, 53

objective function, 430
odd component, 388
odd degree pattern, 336
odd path, 447
odd vertex, 396
2-OPT, 481
2-opt, 481
k-opt, 480
OPTCIRC, 311
OPTFLOW, 301
k-optimal, 480
ε-optimal, 306
optimal circulation, 280, 304
optimal flow, 281, 299, 328
optimal flow problem, 281
optimal matching, 420
optimal pseudoflow, 304
optimal realization, 63
optimal tour, 19, 458
optimization problem, 50
optimum communication spanning tree, 121
optimum requirement spanning tree, 381
optimum requirement tree, 121
OPTMATCH, 420, 583
or, 42
ordered abelian group, 429
orientable, 27
orientation, 25
complete, 25
transitive, 265
oriented 1-factorization, 28–31
originating edge, 202
out-of-kilter algorithm, 298
outdegree, 26
outer vertex, 396

P, 51
painting lemma, 292
parallel class, 231
parallel edges, 13
parallelism, 231
parametric budget problem, 383
parametrized flow problem, 184
parity check extension, 339
partial difference set, 275
partial SDR, 219
partial transversal, 219
partially ordered set, 47, 228–231
partition matroid, 221
path, 5
alternating, 390
augmenting, 155, 351, 390
critical, 73
directed, 26
edge disjoint, 209
Eulerian, 13
even, 447
Hamiltonian, 52, 143
longest, 61
odd, 447
reliable, 506

shortest, 60
vertex disjoint, 209
path algebra, 91
PATHNR, 554
penalty, 466
penalty function, 466
perfect graph, 267
perfect graph theorem, 267
perfect matching, 215, 387
perfect matching polytope, 437
permanent (of a matrix), 225
permanent evaluation problem, 506
permutation matrix, 224
Petersen graph, 24, 216, 548, 579
phase, 173, 199, 322, 393, 424
PIUPDATE, 361
pivot cycle, 354
PIVOTCYCLE, 360
planar graph, 21–25
plane graph, 21
Platonic solids, 22
point, 231
cut, 245
Steiner, 115
vertex, 179
polyhedral combinatorics, 499
polynomial algorithm, 45
polynomial problem, 49
polytope, 432
poset, *see* partially ordered set
positive diagonal, 224
post-optimization, 480
POTENTIAL, 308, 569
potential, 196, 291, 305, 348
minimal, 179
potential difference, 291
predecessor index, 358
preflow, 190
price function, 305
PRIM, 107
primal-dual algorithm, 434
priority, 79
priority queue, 79
problem
t-join, 453
3-SAT, 50, 506
algebraic assignment, 429
assignment, 227, 281
asymmetric travelling salesman, 459, 490
bottleneck, 114, 150
bottleneck assignment, 226, 429
bounded diameter spanning tree, 120
cardinality matching, 419
caterer, 282
Chinese postman, 438, 501
chromatic index, 501
chromatic number, 501
clique, 55, 502
connector, 97
decision, 49
degree constrained spanning tree, 119
directed Hamiltonian cycle, 52
directed Hamiltonian path, 143
discrete metric realization, 502
easy, 45, 49
euclidean Steiner, 115
Euclidean travelling salesman, 477
evaluation, 50
exact perfect matching, 450
geometric Steiner tree, 115
graph partitioning, 503
Hamiltonian cycle, 49, 503
Hamiltonian path, 52
hard, 45
Hitchcock, 328
independent set, 55, 503
induced subgraph, 504
integer linear programming, 431, 504
intractable, 45
isomorphic spanning tree, 119
Kirkman's school girl, 217
knapsack, 61
knight's, 17
Königsberg bridge, 1
length restricted disjoint paths, 503
linear programming, 430
longest cycle, 504
longest path, 53, 504
matroid intersection, 143, 504
max cut, 505
max-flow, 280
maximum leaf spanning tree, 119
metric Steiner network, 116
metric travelling salesman, 459
min cut, 505

minimal cost reliability ratio spanning tree, 120
minimum k-connected subgraph, 505
minimum spanning tree, 505
most uniform spanning tree, 115
network reliability, 114, 506
NP-complete, 46, 51
NP-hard, 53
optimal flow, 281
optimization, 50
optimum communication spanning tree, 121
parametric budget, 383
parametrized flow, 184
permanent evaluation, 506
polynomial, 49
restricted Hamiltonian cycle, 485
restricted perfect matching, 450, 506
satisfiability, 50, 506
shortest cycle, 68, 507
shortest path, 282, 507
shortest total path length spanning tree, 119
spanning tree, 507
stable marriage, 451
Steiner network, 116, 507
Steiner tree, 507
supply and demand, 234–237
transportation, 328
transshipment, 327
travelling salesman, 19, 458, 508
TSP suboptimality, 487
unextendable matching, 508
vertex cover, 53, 508
weighted diameter, 502
weighted matching, 419
zero-one linear programming, 431
problem class, 34
product-optimal matching, 429
program, 33
project evaluation and review technique, 72
project schedule, 72–76
Prüfer code, 10
pseudoflow, 304
ε-optimal, 306
ε-tight, 308
optimal, 304
pseudograph, 13
pseudosymmetric, 26
pseudovertex, 401
PULL, 180
PUSH, 180, 192, 316
admissible, 192, 316
non-saturating, 196, 318
saturating, 196, 318

quasi-inverse, 93
queue, 64

ramification, 42
randomly matchable graph, 394
RANK, 76
rank
in a digraph, 76
in a matroid, 131
lower, 138
upper, 138
rank quotient, 138
rate of growth, 44
reduced cost function, 305, 348
reduced matrix, 490
reduction, transitive, 87
Redéi's theorem, 229
REFINE, 311–322
regular automorphism group, 271
regular bipartite graph, 216
regular graph, 4
RELABEL, 192, 316
admissible, 192, 316
relaxation, 460
s-tree, 463–465
assignment, 461, 497
Lagrange, 470
LP, 465
MST, 462
reliable path, 506
repeat ... **until**, 42
representable, 131
request function, 365
RESIDUAL, 305, 568
residual capacity, 191, 305
residual edge, 191, 305
residual graph, 191, 305
resolution, 4
restricted Hamiltonian cycle, 485
restricted perfect matching, 450, 506
restriction of a matroid, 416

return arc, 280
RHC, *see* restricted hamiltonian cycle
root, 26, 99
RPM, *see* restricted perfect matching problem

SAT, *see* satisfiability problem
3-SAT, 50, 506
satisfiability problem, 50, 506
saturated edge, 154
saturated vertex, 389
saturating PUSH, 196, 318
scatter number, 223
sceleton, 497
schedule
  league, 30–32
  project, 72–76
  tournament, 28–32
  train, 81–84
SDR, 218
  partial, 219
separable graph, 246
separator
  edge, 209
  vertex, 209, 211
set
  closed, 132
  dependent, 131
  feasible, 148
  generating, 132, 135
  independent, 55, 128, 227, 503
  of edges, 2
  partial difference, 275
  partially ordered, 47, 228–231
  stable, 55, 227
set system, 148
  accessible, 148
shortest cycle problem, 68, 507
shortest path, 60
shortest path problem, 282, 507
shortest path tree, *see* directed Hamiltonian path problem, *see* SP-tree
sink, 153, 344
six color theorem, 276
slack, 73
source, 153, 344
SP-tree, 69, 90
space complexity, 44
span, 132
  linear, 437
span operator, 132
spanning arborescence, 69, 121–125, 244
spanning Eulerian multigraph, 472
spanning forest
  minimal, 104
spanning subgraph, 3
spanning tree, 65, 101, 484, 507
  maximal, 113–115
  minimal, 104–112, 463, 472
spanning tree problem, 507
spanning tree with restrictions, 119–121
sparse, 45
Sperner's lemma, 230
SPTREE, 526
SRG, *see* strongly regular graph
stable marriage problem, 451
stable matching, 451
stable roommates problem, 451
stable set, 55, 227
stack, 250
star, 11
start vertex, 5, 25
state, 275
steepest descent, 480
STEINER, 117
Steiner network problem, 116, 507
Steiner point, 115
Steiner points, 116
Steiner ratio conjecture, 116
Steiner tree, 115–118, 507
  minimal, 116, 477
Steiner tree problem, 507
stem, 406
step (in an algorithm), 44
strong component, 253–257
strong duality theorem, 433
strong exchange axiom, 149
strong extensibility axiom, 151
strong perfect graph theorem, 268
STRONGCOMP, 256
strongly connected, 26, 27, 253–257
strongly polynomial, 298
strongly regular graph, 5, 39
subdivision, 23
subfamily
  critical, 219
subgradient, 470

subgradient optimization, 466
subgraph, 3
  equality, 421
  even, 332
  induced, 3
  odd, 332
  spanning, 3
submodular, 131
suboptimal, 487
subtour elimination constraints, 465
supply, 235, 344
supply and demand problem, 234–237
supply and demand theorem, 235
support, 159
Sylvester's conjecture, 234
symmetric bipartite graph, 416
symmetric digraph, 190
symmetric matching, 416
symmetric network, 363
system of distinct representatives, *see* SDR
system of representatives, 218
  common, 222

tail, 25
tail-partition matroid, 129
term rank, 223
termination, 34
theorem
  1-factor, 388
  augmenting path, 155, 391
  basis completion, 132
  circulation, 288
  decomposition, 224
  five color, 278
  four color, 278
  harem, 223
  integral flow, 156
  marriage, 218
  matrix tree, 101
  max-flow min-cut, 156
  perfect graph, 267
  strong duality, 433
  supply and demand, 235
  total dual integrality, 434
theorem of
  Baranyai, 231
  Birkhoff, 225
  Cauchy and Binet, 102
  Euler, 13
  Gale and Ryser, 236
  Phillip Hall, 215
  Brooks, 263
  Cook, 52
  Dilworth, 228
  Ford and Fulkerson, 155–156
  Kuratowski, 23
  König, 214
  Menger, 209–212
  Redéi, 229
  Robbins, 27
  Stern and Lenz, 273
  Tutte, 388
  Vizing, 268
  Wagner, 24
  Whitney, 212, 239
thread index, 359
TIGHT, 311, 570
ε-tight, 308
time complexity, 44
time cycle, 81
topological sorting, 47
TOPSORT, 48
total dual integrality theorem, 434
totally dual integral, 434
totally unimodular, 101
tour, 19, 458
  k-optimal, 480
  Euler, 13
  optimal, 19, 458
  suboptimal, 487
tournament, 229
  schedules, 28–32
TRACE, 40, 43
trail, 5, 25
  closed, 5
  directed, 26, 27
  Eulerian, 13
train schedule, 81–84
trajectory, 202
  long, 203
transitive closure, 85
transitive digraph, 85
transitive orientation, 265
transitive reduction, 87
transportation problem, 328
transshipment node, 235, 344
transshipment problem, 327

transversal, 218
  common, 222
  partial, 219
transversal index, 219
transversal matroid, 220
travelling salesman problem, 19, *see* TSP
tree, 8, 26, 97
  s-, 463
  alternating, 395
  cut, 379–383
  directed, *see* arborescence
  dominant requirement, 373
  equivalent flow, 365
  maximal spanning, 113–115
  minimal spanning, 104–112, 463, 472, 505
  rooted, 99
  shortest path, 69
  SP-, 69, 90
  spanning, 65, 101, 484, 507
    with restrictions, 119–121
  Steiner, 115–118, 507
    minimal, 116, 477
  uniform, 373
s-tree relaxation, 463–465
tree algorithm, 472
tree edge, 244, 252
tree graph, 112
tree indices, 358
tree solution, 346
tree structure, 348
  admissible, 348
  degenerate, 354
  nondegenerate, 355
  optimal, 348
  strongly admissible, 355
triangle factor, 216
triangle inequality, 62
triangular graph, 5
triangulated, 266
TSP, 508
  Δ, 459
  asymmetric, 459
  Euclidean, 477, 482
  metric, 459
TSP suboptimality, 487

underlying graph, 25
underlying multigraph, 25
unextendable matching, 213, 508
unextendable matching problem, 508
unicyclic graph, 98
uniform graph, 373
uniform matroid, 129
uniform tree, 373
upper capacity, 279, 344
upper rank, 138

valid labelling, 191
value (of a flow), 154
van der Waerden's conjecture, 225
VC, 53, *see* vertex cover problem
vectorial matroid, 131
vertex, 2, 25
  accessible, 26
  active, 192, 315
  current, 320
  end, 2, 5, 25
  even, 396
  exposed, 389
  inner, 396
  isolated, 6
  odd, 396
  of a polytope, 432
  outer, 396
  pseudo-, 401
  saturated, 389
  start, 5, 25
vertex cover, 53, 508
vertex cover problem, 53, 508
vertex disjoint paths, 209, 503
vertex separator, 209, 211
vertex set, 2, 25
Vizing's theorem, 268
void edge, 154

walk, 5
  closed, 5
weakly polynomial, 298
weight, 19, 104, 128, 331, 420
weight enumerator, 333
weighted matching problem, 419
while ... **do**, 42
Whitney's theorem, 212, 239
width (of a matrix), 224
word, 40

zero-one linear program, 431

zero-one linear programming problem, 431
zero-one matrix, 225, 227, 237
zero-one-flow, 185
zero-one-network, 185
ZOLP, *see* zero-one linear programming problem